AVR单片机快速入门

徐玮　沈建良　徐苏　安康◎等编著

实例丰富
图文并茂
视频演示

机械工业出版社
CHINA MACHINE PRESS





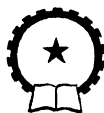
赠1CD

VISIT…

LANZAROTE
Caliente.COM

AVR 单片机快速入门

徐　玮　沈建良　徐　苏　安　康　等编著



机 械 工 业 出 版 社

本书是以目前最为流行的AVR系列单片机为主体，使用C语言来进行描述。本书共分为五部分内容：单片机基础知识、C程序设计知识、单片机入门基础实例、单片机高级应用实例、配套学习套件的使用说明。本书采用理论与实践相结合的方式进行讲解，避免了传统教科书给人枯燥、乏味的感觉。讲解风格通俗易懂，条理清晰，实例丰富，图文并茂，并带视频演示，即使是没有接触过单片机的读者，也可以通过本书的学习快速跨入单片机世界的大门。

作者为本书的出版开发了相应的单片机学习套件，以方便读者进行学习，同时以大量实例照片和视频录像记录了实验的全过程及现象，更加激发了读者对单片机的兴趣爱好，读者也可以在网站http://www.hificat.com进行单片机知识的学习与交流。

本书的配套光盘含有所有实验的源程序代码、一些常用的电子工具软件、芯片资料、实验过程照片以及实验演示视频录像。因此，有了本书，读者获得的是教程和学习平台的结合，不仅可以用来学习，还可以供工厂、企业的工程技术人员进行产品研发时参考。

本书适合从事单片机应用研发的技术人员和高校相关专业师生阅读。

图书在版编目（CIP）数据

AVR单片机快速入门/徐玮等编著. —北京：机械工业出版社，2011.11

ISBN 978-7-111-36320-0

Ⅰ.①A… Ⅱ.①徐… Ⅲ.①单片微型计算机 Ⅳ.①TP368.1

中国版本图书馆CIP数据核字（2011）第224041号

机械工业出版社（北京市百万庄大街22号 邮政编码100037）
策划编辑：林春泉 责任编辑：闫洪庆 版式设计：霍永明
责任校对：刘志文 封面设计：路恩中 责任印制：杨 曦
保定市中画美凯印刷有限公司印刷
2012年2月第1版第1次印刷184mm×260mm·24.25印张·614千字
0001—3000册
标准书号：ISBN 978-7-111-36320-0
ISBN 978-7-89433-236-3（光盘）
定价：68.00元（含1CD）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务	网络服务
社服务中心：（010）88361066	门户网：http://www.cmpbook.com
销售一部：（010）68326294	教材网：http://www.cmpedu.com
销售二部：（010）88379649	
读者购书热线：（010）88379203	**封面无防伪标均为盗版**

前　　言

当今世界科学技术飞速发展，以前需要花费大量的时间和精力来搭建一个模拟电路，同时大量的元器件也增加了产品的成本；而现在只需要一块小小的单片机芯片，写入相应功能的程序，便可以代替以前分立元器件组成的电路了。相信读者掌握了单片机技术后，无论是在产品开发还是工作上，都会有意想不到的惊喜。

本书作者着眼于快速入门、通俗易懂、趣味学习、学以致用的指导思想。以理论与实践相结合为主线，通过通俗易懂的讲解，丰富的实例，图文并茂的编排，以及配套光盘中各程序实例的视频演示录像，使读者能够轻松地掌握单片机的基础知识，并使读者具有初步开发、设计单片机产品的能力。相信读者通过本书的学习，即使是一位单片机的“门外汉”，也能运用单片机的知识来解决一些实际问题，将知识转化为生产力。

本书共分为五部分内容：单片机基础知识、C程序设计知识、单片机入门基础实例、单片机高级应用实例、配套学习套件的使用说明。

单片机基础知识：介绍单片机的发展历史，揭开它的神秘之处。初学者最关心的一个实际问题是单片机到底能够做哪些事？这也是我们要学习单片机技术的理由之一。当我们明确了学习目标后，就需要做好学习实践平台的准备，在本书中将一一为读者进行讲解，如单片机学习的有效方法和途径，单片机的内部结构、引脚定义、存储器、寄存器、定时/计数器、中断系统、串行通信等相关知识，让读者对单片机有实质性的了解。

C程序设计知识：经常会有人问，使用单片机用C语言好，还是用汇编语言好。这两种语言都有各自的特点。汇编语言的优点是比较灵活，但程序不易理解，对产品的升级、维护不太有利；而C语言已有了非常丰富的库函数供用户使用，因为它是高级语言，程序代码的编写也非常人性化，易于阅读、理解，C语言已经成为一门在整个计算机业都普遍应用的语言。因此，本书也是以C语言来进行描述的，我们将会介绍C语言的数据类型、运算符、表达式，分支与循环控制语句，编译预处理与位运算，数组与函数，指针、结构体与共用体等知识，使读者具有C语言程序设计的能力。

单片机入门基础实例：前面几章讲的都是理论知识内容，由于单片机是一门实践性非常强的技术，即使有再多的理论基础，也必须通过较多的实践操作才能真正学好这门技术。因此，在这部分章节中，我们将为读者先引入一系列具有趣味性、简单易懂的基础实验实例，如点亮一个发光管，流水灯控制，按键、蜂鸣器、数码管、继电器的操作和使用，串行通信等。在此，我们暂时不求技术深，只求让读者明白单片机到底是如何来实现我们所需要的特定功能的，我们又是如何通过软件的程序来最终从硬件功能上反映出来的。

单片机高级应用实例：熟悉了前面介绍的基础实例，想必读者已经对单片机有了一定程度的认识，知道实现怎么样的功能，应该写怎么样的程序。在这部分内容中，我们将为读者做一些单片机高级应用实例的介绍，让读者从单片机知识学习的水平升华到产

品开发的程度。有液晶显示、步进电动机控制、I^2C 总线原理、数字温度传感器应用、无线通信控制、SD 卡读写、PWM 应用、LED 点阵显示屏、红外线遥控的软件解码、模-数转换器应用实例、DS1302 时钟芯片的应用等。看完这部分内容，相信读者已经跨入了单片机世界的大门，并具有初步的产品开发能力了，剩下的是靠时间来积累实践经验了。

配套学习套件的使用说明：详细地介绍了与本书相配套的 AVR 单片机综合学习系统的原理与使用方法。“AVR 单片机综合学习系统”是综合多年经验开发出的多功能 AVR 单片机平台。集成常用的单片机外围硬件，提供 ISP 程序下载和 JTAG 仿真接口。系统附带众多 C 语言例子程序，可以让读者在最短的时间内，全面地了解掌握单片机编程技术，特别适合于单片机初学者、大中专院校学生、单片机工程师及实验室选用。这部分内容详细地介绍了如何使用 AVR 单片机综合学习系统来进行程序编写、开发、设计的全过程。

为了方便广大读者的学习交流，读者可以访问网站 http://www.hificat.com 来做互相交流。同时，如果读者对本书中所用到的学习器材、设备有兴趣的话，也可以访问我们的网站查询购买方法。当然，更新更详细的学习资料及内容，也都会定期地放到网上供读者使用。

参加本书编写的还有徐金林、卢水英、邵磊、邵晶晶、彭敏芳、戴婧、魏巍、韩珈骏、蔡东琦、孙燕、沈媛媛、徐富军、庄建清、王琴、杨青、杨丹枫、杨莺、许敏、卢剑、金向红等。我们衷心地希望本书能够对从事单片机技术工作的读者有所帮助。

由于本书程序实例和演示图表比较多，作者水平有限，难免会有错误与不妥之处，请广大读者批评指正。

作　者

2011 年 10 月

目　　录

前言
第 1 章　单片机嵌入式系统概述 ………… 1
1.1　嵌入式系统简介 ………………………… 1
1.1.1　嵌入式计算机 ……………………… 1
1.1.2　单片机嵌入式系统 ………………… 2
1.1.3　单片机的发展历史 ………………… 2
1.1.4　单片机的发展趋势 ………………… 3
1.2　单片机嵌入式系统的结构与应用领域 … 4
1.2.1　单片机嵌入式系统的结构 ………… 4
1.2.2　单片机嵌入式系统的应用领域 …… 6
1.3　AVR 单片机简介 ………………………… 7
1.3.1　ATMEL 公司的单片机简介 ………… 7
1.3.2　AVR 单片机的主要特点 …………… 8
1.3.3　AVR 单片机最小系统 …………… 10
第 2 章　AVR 单片机的基本结构 ……… 11
2.1　单片机的基本组成 ……………………… 11
2.1.1　单片机的基本组成结构 …………… 11
2.1.2　单片机的基本单元与作用 ………… 11
2.2　ATmega16 单片机的组成 ……………… 14
2.2.1　AVR 单片机的内核结构 ………… 14
2.2.2　ATmega16 的特点 ………………… 16
2.2.3　ATmega16 的外部引脚与封装 …… 17
2.3　ATmega16 单片机的内部结构 ………… 18
2.3.1　中央处理器 ………………………… 18
2.3.2　系统时钟部件 ……………………… 20
2.3.3　CPU 的工作时序 …………………… 22
2.3.4　存储器 ……………………………… 22
2.3.5　I/O 口 ……………………………… 23
2.4　存储器结构和地址空间 ………………… 23
2.4.1　支持 ISP 的 Flash 程序存储器 …… 23
2.4.2　SRAM 数据存储器空间 …………… 24
2.4.3　内部 EEPROM 存储器 …………… 24
2.5　通用寄存器组与 I/O 寄存器 …………… 25
2.5.1　通用寄存器组 ……………………… 25
2.5.2　I/O 寄存器 ………………………… 26
2.5.3　状态寄存器和堆栈指针寄存器 …… 27
2.6　ATmega16 单片机的工作状态 ………… 29
2.6.1　AVR 单片机最小系统 …………… 30
2.6.2　AVR 单片机的复位源和复位方式 …………………………………… 31
2.6.3　对 AVR 单片机的编程下载 ……… 34
2.6.4　ATmega16 的熔丝位 ……………… 35
2.6.5　AVR 单片机的工作状态 ………… 37
2.6.6　支持 ISP 编程的最小系统设计 …… 38
2.7　AVR 单片机内部资源的扩展和删减 … 40
第 3 章　AVR 单片机开发工具安装及开发环境的使用 …………………… 41
3.1　AVR Studio 集成开发环境简介及其安装 ………………………………………… 41
3.2　AVR Studio 集成开发环境的使用 ……… 43
3.2.1　建立一个新的工程项目管理文件 …………………………………… 43
3.2.2　汇编源文件的建立 ………………… 44
3.2.3　汇编源文件的编译 ………………… 45
3.3　ICCAVR 集成开发环境简介 …………… 46
3.3.1　ICCAVR 编译器的安装 …………… 46
3.3.2　ICCAVR 中的文件类型及其扩展名 ………………………………… 48
3.3.3　ICCAVR 的附注和扩充 …………… 49
3.3.4　ICCAVR 的代码转换 ……………… 50
3.4　ICCAVR 向导 ……………………………… 50
3.5　ICCAVR 的 IDE 环境 …………………… 52
3.6　菜单解释 …………………………………… 53
3.7　C 库函数与启动文件 …………………… 56
3.8　访问 AVR 单片机硬件的编程 ………… 63
3.9　C 语言的运行结构 ……………………… 70
3.10　其他主流 AVR 单片机开发环境简介 ……………………………………… 72
3.10.1　GCCAVR 开发环境 ……………… 72
3.10.2　CodeVision AVR 集成开发环境 … 72
3.10.3　IAR 集成开发环境 ……………… 72
第 4 章　C 语言概论、数据类型、运算符与表达式 ………………………… 74
4.1　C 语言概论 ……………………………… 74
4.1.1　C 语言的发展过程 ………………… 74
4.1.2　C 语言的特点 ……………………… 74

4.1.3　C 源程序的结构特点 …… 74
4.1.4　C 语言的字符集 …… 75
4.1.5　C 语言的词汇 …… 76
4.2　数据类型、运算符与表达式 …… 77
4.2.1　C 语言的数据类型 …… 77
4.2.2　算术运算符和算术表达式 …… 84
4.2.3　关系运算符和表达式 …… 88
4.2.4　逻辑运算符和表达式 …… 89
第 5 章　分支与循环控制 …… 92
5.1　if 语句 …… 92
5.1.1　程序的 3 种基本结构 …… 92
5.1.2　if 语句的 3 种形式 …… 92
5.1.3　if 语句的嵌套 …… 96
5.2　条件运算符和条件表达式 …… 98
5.3　switch 语句 …… 99
5.4　循环控制 …… 102
5.4.1　概述 …… 102
5.4.2　goto 语句和 if 语句构成循环 …… 103
5.4.3　while 语句 …… 103
5.4.4　do-while 语句 …… 105
5.4.5　for 语句 …… 106
5.4.6　循环的嵌套 …… 108
5.4.7　break 和 continue 语句 …… 109
第 6 章　编译预处理与位运算 …… 112
6.1　概述 …… 112
6.2　宏定义 …… 112
6.2.1　不带参数的宏定义 …… 112
6.2.2　带参数的宏定义 …… 114
6.3　文件包含 …… 115
6.4　条件编译 …… 116
6.5　位操作运算符 …… 118
第 7 章　数组与函数 …… 121
7.1　一维数组的定义和引用 …… 121
7.1.1　一维数组的定义方式 …… 121
7.1.2　一维数组元素的引用 …… 122
7.1.3　一维数组的初始化 …… 124
7.1.4　一维数组程序举例 …… 124
7.2　二维数组的定义和引用 …… 125
7.2.1　二维数组的定义 …… 125
7.2.2　二维数组元素的引用 …… 126
7.2.3　二维数组的初始化 …… 127
7.3　字符数组 …… 128
7.3.1　字符数组的定义 …… 128
7.3.2　字符数组的初始化 …… 128
7.3.3　字符数组的引用 …… 129
7.3.4　字符串和字符串结束标志 …… 129
7.4　函数概述 …… 129
7.4.1　函数定义的一般形式 …… 130
7.4.2　函数的参数和函数的值 …… 131
7.4.3　函数的返回值 …… 132
7.4.4　函数的调用 …… 132
7.4.5　被调用函数的声明和函数原型 …… 132
7.4.6　函数的嵌套调用 …… 133
7.4.7　函数的递归调用 …… 134
7.4.8　数组作为函数参数 …… 135
7.5　局部变量和全局变量 …… 137
7.5.1　局部变量 …… 138
7.5.2　全局变量 …… 139
第 8 章　指针、结构体与共用体 …… 141
8.1　指针和地址 …… 141
8.2　指针变量和指针运算符 …… 141
8.3　指针与函数参数 …… 145
8.4　指针、数组和字符串指针 …… 146
8.5　指针数组 …… 149
8.6　多级指针 …… 151
8.7　返回指针的函数 …… 152
8.8　函数指针 …… 153
8.9　结构与联合 …… 154
8.9.1　结构的定义 …… 154
8.9.2　结构数组 …… 156
8.9.3　结构与函数 …… 157
8.9.4　结构的初始化 …… 159
8.9.5　联合 …… 159
第 9 章　AVR 开发套件快速入门 …… 161
9.1　AVR 单片机实验系统简介 …… 161
9.2　建立第一个项目（软件操作指南） …… 164
9.3　AVR 单片机综合学习系统芯片烧写操作指南 …… 167
9.4　AVR ATmega16 单片机引脚说明 …… 170
第 10 章　ATmega16 基础实例 …… 173
10.1　发光二极管闪动实验 …… 173
10.1.1　实例功能 …… 173
10.1.2　器件与原理 …… 173
10.1.3　硬件电路 …… 174
10.1.4　程序设计 …… 175
10.2　流水灯实验 …… 176
10.3　按键实验 …… 180
10.3.1　实例功能 …… 180

10.3.2　器件与原理 …… 181
10.3.3　程序设计 …… 182
10.4　蜂鸣器实验 …… 185
10.4.1　实例功能 …… 185
10.4.2　器件与原理 …… 185
10.4.3　硬件电路 …… 185
10.4.4　程序设计 …… 186
10.5　继电器实验 …… 188
10.5.1　实例功能 …… 188
10.5.2　器件与原理 …… 188
10.5.3　硬件电路 …… 189
10.5.4　程序设计 …… 189
10.6　数码管实验 …… 191
10.6.1　实例功能 …… 191
10.6.2　器件与原理 …… 191
10.6.3　硬件电路 …… 194
10.6.4　程序设计 …… 194
10.7　串行口实验 …… 196
10.7.1　实例功能 …… 197
10.7.2　硬件电路 …… 200
10.7.3　程序设计 …… 202
第11章　ATmega16高级应用实例 …… 206
11.1　矩阵键盘应用实例 …… 206
11.1.1　矩阵键盘简介 …… 206
11.1.2　矩阵键盘的工作原理 …… 206
11.1.3　矩阵键盘软硬件设计实例 …… 206
11.2　步进电动机应用实例 …… 212
11.2.1　步进电动机简介 …… 212
11.2.2　步进电动机的控制 …… 219
11.2.3　步进电动机的应用设计 …… 221
11.3　DS18B20单总线数字温度传感器应用实例 …… 224
11.3.1　单总线技术简介 …… 224
11.3.2　DS18B20单总线温度传感器简介 …… 225
11.3.3　DS18B20软硬件设计 …… 230
11.4　I²C总线应用实例 …… 236
11.4.1　I²C串行总线简介 …… 237
11.4.2　I²C总线器件工作原理及时序 …… 238
11.4.3　AT24C系列存储器的软硬件设计实例 …… 242
11.5　93CXX系列存储器应用实例 …… 251
11.5.1　SPI总线简介 …… 251
11.5.2　93C46存储器的软硬件设计实例 …… 254
11.6　DS1302时钟芯片应用实例 …… 265
11.6.1　实时时钟（RTC）简介 …… 265
11.6.2　DS1302的软硬件设计实例 …… 268
11.7　ADC应用实例 …… 277
11.7.1　ATmega16片内ADC内部寄存器 …… 277
11.7.2　ADC软硬件设计实例 …… 280
11.8　1602字符型LCD应用实例 …… 284
11.8.1　液晶显示简介 …… 284
11.8.2　1602字符型LCD简介 …… 285
11.8.3　1602LCD的软硬件设计实例 …… 290
11.9　12864点阵型LCD应用实例 …… 294
11.9.1　点阵LCD的显示原理 …… 295
11.9.2　12864点阵型LCD简介 …… 295
11.9.3　12864点阵型LCD软硬件设计实例 …… 300
11.10　红外遥控软件解码应用实例 …… 312
11.10.1　红外遥控概述 …… 312
11.10.2　μPD6121红外接收的软件解码应用实例 …… 318
11.10.3　μPD6121解码应用设计 …… 318
11.11　无线通信模块应用实例 …… 328
11.11.1　无线通信模块原理与分类 …… 329
11.11.2　无线通信模块主要技术指标 …… 329
11.11.3　PT2262/PT2272无线模块简介 …… 330
11.11.4　无线通信模块的软硬件设计应用 …… 334
11.12　PWM应用实例 …… 338
11.12.1　PWM的特点 …… 338
11.12.2　ATmega16内部PWM简介 …… 339
11.12.3　基于ATmega16的PWM应用设计 …… 346
11.13　SD卡读写实例 …… 349
11.13.1　SD卡简介 …… 349
11.13.2　SD卡读写应用实例 …… 362
11.14　LED点阵显示屏的应用实例 …… 374
11.14.1　LED点阵的种类及结构 …… 374
11.14.2　8×8单色点阵LED的工作原理 …… 374
11.14.3　LED点阵显示屏系统设计 …… 375
参考文献 …… 379

第 1 章　单片机嵌入式系统概述

1.1　嵌入式系统简介

嵌入式系统是以应用为中心，以计算机技术为基础，并且软硬件可裁剪，适用于应用系统对功能、可靠性、成本、体积、功耗有严格要求的专用计算机系统。它一般由嵌入式微处理器、外围硬件设备、嵌入式操作系统以及用户的应用程序等 4 个部分组成，用于实现对其他设备的控制、监视或管理等功能。图 1-1 所示为典型嵌入式系统框图。

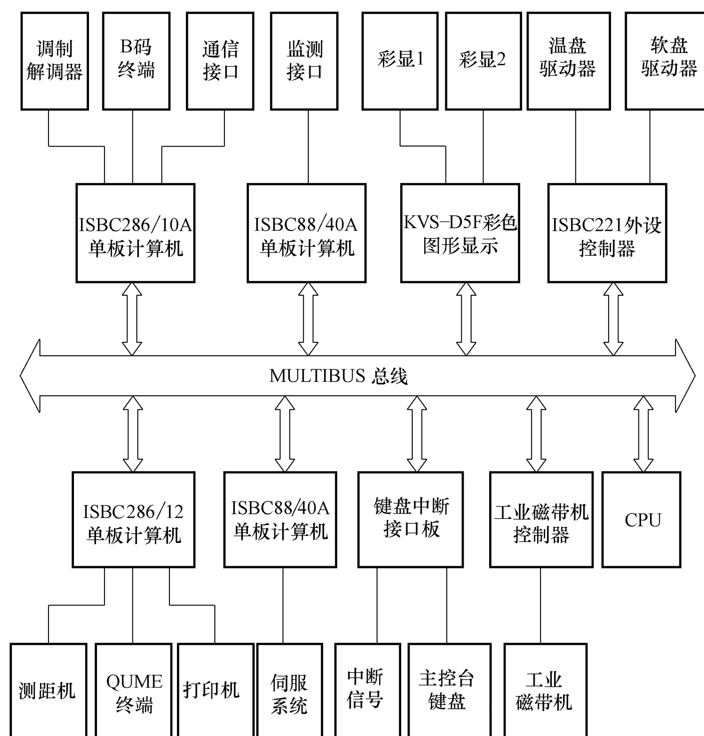


图 1-1　典型嵌入式系统框图

1.1.1　嵌入式计算机

嵌入式计算机在应用数量上远远超过了各种通用计算机，一台通用计算机的外部设备中就包含了 5 ~ 10 个嵌入式微处理器，键盘、鼠标、软驱、硬盘、显示卡、显示器、调制解调器、网卡、声卡、打印机、扫描仪、数字相机、USB 集线器等均是由嵌入式处理器控制的。在制造工业、过程控制、通信、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等方面均是嵌入式计算机的应用领域。嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物，这一点就决定了它必然是一个技术密

集、资金密集、高度分散、不断创新的知识集成系统。

1.1.2　单片机嵌入式系统

嵌入式系统一般指非 PC 系统，它包括硬件和软件两部分。硬件包括处理器/微处理器、存储器、外设器件、I/O 端口和图形控制器等。软件部分包括操作系统（OS）软件（要求实时和多任务操作）和应用程序编程。有时设计人员把这两种软件组合在一起。应用程序控制着系统的运行和行为，而操作系统控制着应用程序编程与硬件的交互作用。

嵌入式系统的核心是嵌入式微处理器。嵌入式微处理器一般具备以下 4 个特点：

1）对实时多任务有很强的支持能力，能完成多任务，并且有较短的中断响应时间，从而使内部的代码和实时内核心的执行时间减少到最低限度。

2）具有功能很强的存储区保护功能。这是由于嵌入式系统的软件结构已模块化，而为了避免在软件模块之间出现错误的交叉作用，需要设计强大的存储区保护功能，同时也有利于软件诊断。

3）可扩展的处理器结构，能最迅速地开发出满足应用的最高性能的嵌入式微处理器。

4）嵌入式微处理器必须功耗很低，尤其是用于便携式的无线及移动的计算和通信设备中靠电池供电的嵌入式系统更是如此，如需要功耗只有 mW 甚至 μW 级。

由此可知，单片机与嵌入式系统是紧密相连、密不可分的。单片机系统与嵌入式系统相比具有开发周期短、成本低、使用灵活等特点，在中小系统设计中具有较强的优势。

1.1.3　单片机的发展历史

单片机诞生于20 世纪 70 年代末，经历了 SCM、MCU、SoC 三大阶段。①SCM 即单片微型计算机（Single Chip Microcomputer）阶段，主要是寻求最佳的单片形态嵌入式系统的最佳体系结构。“创新模式”获得成功，奠定了 SCM 与通用计算机完全不同的发展道路。在开创嵌入式系统独立发展的道路上，Intel 公司功不可没。②MCU 即微控制器（Micro Controller Unit）阶段，主要的技术发展方向是：不断扩展满足嵌入式应用时对象系统要求的各种外围电路与接口电路，突显其对象的智能化控制能力。它所涉及的领域都与对象系统相关，因此发展 MCU 的重任不可避免地落在电气、电子技术厂商。从这一角度来看，Intel 公司逐渐淡出 MCU 的发展也有其客观因素。在发展 MCU 方面，最著名的厂商当数 Philips 公司。Philips 公司以其在嵌入式应用方面的巨大优势，将 MCS-51 从单片微型计算机迅速发展到微控制器。因此，当回顾嵌入式系统发展道路时，不要忘记 Intel 公司和 Philips 公司的历史功绩。③单片机是嵌入式系统的独立发展之路，向 MCU 阶段发展的重要因素。就是寻求应用系统在芯片上的最大化解决。因此，专用单片机的发展自然形成了 SoC（System on a Chip，系统级芯片）化趋势。随着微电子技术、IC 设计、EDA 工具的发展，基于 SoC 的单片机应用系统设计会有较大的发展。因此，对单片机的理解可以从单片微型计算机、单片微控制器延伸到单片应用系统。

单片机作为微型计算机的一个重要分支，应用面很广，发展很快。自单片机诞生至今，已发展为上百种系列的近千个机种。如果将 8 位单片机的推出作为起点，那么单片机的发展历史大致可分为以下几个阶段：

1）第一阶段（1976—1978）：单片机的探索阶段。以 Intel 公司的 MCS – 48 为代表。MCS – 48 的推出是在工控领域的探索，参与这一探索的公司还有 Motorola 、Zilog 等，都取

得了满意的效果。这就是 SCM 的诞生年代，“单片机”一词即由此而来。

2）第二阶段（1978—1982）：单片机的完善阶段。Intel 公司在 MCS - 48 基础上推出了完善的、典型的单片机系列 MCS - 51。它在外部总线完善、外围功能单元的集中管理、位地址空间及位操作方式、丰富和完善指令系统等几个方面奠定了典型的通用总线型单片机体系结构。

3）第三阶段（1982—1990）：8 位单片机的巩固发展及 16 位单片机的推出阶段，也是单片机向微控制器发展的阶段。Intel 公司推出的 MCS-96 系列单片机，将一些用于测控系统的 A/D 转换器、程序运行监视器、脉宽调制器等纳入片中，体现了单片机的微控制器特征。随着 MCS - 51 系列的广泛应用，许多电气厂商竞相使用 80C51 为内核，将许多测控系统中使用的电路技术、接口技术、多通道 A/D 转换部件、可靠性技术等应用到单片机中，增强了外围电路功能，强化了智能控制的特征。

4）第四阶段（1990—至今）：微控制器的全面发展阶段。随着单片机在各个领域全面深入地发展和应用，出现了高速、大寻址范围、强运算能力的 8 位/16 位/32 位通用型单片机，以及小型廉价的专用型单片机。

1.1.4　单片机的发展趋势

目前，单片机正朝着高性能和多品种方向的发展，也就是进一步向着 CMOS（Complementary Metal-Oxide-Semiconductor，互补金属物氧化物半导体）化、低功耗、小体积、大容量、高性能、低价格和外围电路内装化等几个方面发展。单片机的主要发展趋势为：

1）CMOS 化：近年来，由于 CHMOS（互补金属氧化物 HMOS）技术的进步，大大地促进了单片机的 CMOS 化。CMOS 芯片除了低功耗特性之外，还具有功耗的可控性，使单片机可以工作在功耗精细管理状态。这也是今后以 80C51 取代 8051 为标准 MCU 芯片的原因。单片机芯片多数是采用 CMOS（金属栅氧化物）半导体工艺生产。CMOS 电路的特点是低功耗、高密度、低速度、低价格。采用双极型半导体工艺的 TTL 电路速度快，但功耗和芯片面积较大。随着技术和工艺水平的提高，又出现了 HMOS（高性能金属氧化物半导体）和 CHMOS 工艺。CHMOS 工艺、CMOS 工艺和 HMOS 工艺的结合，具有低功耗的特点。目前，生产的 CHMOS 电路已达到 LSTTL（Low-power Schottky TTL，低功耗肖特基 TTL）的速度，传输延迟时间小于 2ns，它的综合优势已优于 TTL（Transistor-Transistor-Logic，晶体管-晶体管逻辑电路）电路。因而，在单片机领域 CMOS 正在逐渐取代 TTL 电路。

2）低功耗化：单片机的功耗已降到 mA 级甚至 1μA 以下，使用电压在 3～6V 之间，完全适应电池工作。低功耗还带来了产品的高可靠性、高抗干扰能力以及产品的便携化。

3）低电压化：几乎所有的单片机都有 WAIT、STOP 等省电运行方式。允许使用的电压范围越来越宽，一般在 3～6V 范围内工作。低电压供电的单片机电源下限已可达 1～2V。目前采用 0.8V 供电的单片机已经问世。

4）低噪声与高可靠性：为提高单片机的抗电磁干扰能力，使产品能适应恶劣的工作环境，满足电磁兼容性方面更高标准的要求，各单片机厂商在单片机内部电路中都采用了新的技术措施。

5）大容量化：以往单片机内的 ROM（Read Only Memory，只读存储器）为 1～4KB，RAM 为 64～128B。但在需要复杂控制的场合，该存储容量是不够的，必须进行外接扩充。为了适应这种领域的要求，必须运用新的工艺，使片内存储器大容量化。目前，单片机内

ROM 最大可达 64KB，RAM 最大为 2KB。

6）高性能化：主要是指进一步改进 CPU 的性能，加快指令运算的速度和提高系统控制的可靠性。采用精简指令集计算机（Reduced Instruction Set Computer，RISC）结构和流水线技术，可以大幅度提高运行速度。现指令速度最高者已达 100MIPS（Million Instruction Per Second，即每秒百万条指令），并加强了位处理功能、中断和定时控制功能。这类单片机的运算速度比标准的单片机高出 10 倍以上。由于这类单片机有极高的指令速度，就可以用软件模拟其 I/O 功能，由此引入了虚拟外设的新概念。

7）小容量、低价格化：与上述相反，以 4 位、8 位机为中心的小容量、低价格化也是发展动向之一。这类单片机的用途是把以往用数字逻辑集成电路组成的控制电路单片化，可广泛用于家电产品。

8）外围电路内装化：这也是单片机发展的主要方向。随着集成度的不断提高，有可能把众多的各种外围功能器件集成在片内。除了一般必须具有的 CPU、ROM、RAM、定时器/计数器等以外，片内集成的部件还有 A/D 转换器、DMA（直接内存存取）控制器、声音发生器、监视定时器、液晶显示驱动器、彩色电视机和录像机用的锁相电路等。

9）串行扩展技术：在很长一段时间里，通用型单片机通过三总线结构扩展外围器件成为单片机应用的主流结构。随着低价位 OTP（One Time Programable，一次性可编程）及各种类型片内程序存储器的发展，加之外围接口不断进入片内，推动了单片机“单片”应用结构的发展。特别是 IC、SPI 等串行总线的引入，可以使单片机的引脚设计得更少，单片机系统结构更加简化及规范化。随着半导体集成工艺的不断发展，单片机的集成度将更高、体积将更小、功能将更强。在单片机家族中，80C51 系列是其中的佼佼者，加之 Intel 公司将其 MCS-51 系列中的 80C51 内核使用权以专利互换或出售的形式转让给全世界许多著名 IC 制造厂商，如 Philips、NEC、ATMEL、AMD、华邦等，这些公司都在保持与 80C51 单片机兼容的基础上改善了 80C51 的许多特性。这样，80C51 就变成有众多制造厂商支持的、发展出上百个品种的大家族，现统称为 80C51 系列。80C51 单片机已成为单片机发展的主流。专家认为，虽然世界上的 MCU 品种繁多，功能各异，开发装置也互不兼容，但是客观发展表明，80C51 可能最终形成事实上的标准 MCU 芯片。

我国开始使用单片机是在 1982 年，短短 5 年时间里发展极为迅速。1986 年在上海召开了全国首届单片机开发与应用交流会，有的地区还成立了单片微型计算机应用协会，那是全国形成的第一次高潮。目前，单片机应用技术飞速发展，我们上因特网输入一个“单片机”进行搜索，将会看到上万个介绍单片机的网站，这还不包括国外的。与它相应的专业杂志现在也有很多，比如由单片机界的资深专家何立民主编的《单片机与嵌入式系统应用》杂志现已风靡电子界，91student. com（中国研究生人才网）曾在北京、上海、广州等大城市所做的一次专业人才需求报告中显示，单片机人才的需求量位居第一。

1.2　单片机嵌入式系统的结构与应用领域

1.2.1　单片机嵌入式系统的结构

仅由一片单片机芯片是不能构成一个应用系统的。系统的核心控制芯片往往还需要与一些外围芯片、器件和控制电路机构有机地连接在一起，才能构成一个实际的单片机系统，进

而再嵌入到应用对象的环境体系中，作为其中的核心智能化控制单元而构成典型的单片嵌入式应用系统，如洗衣机、电视机、空调、智能仪器、智能仪表等。

单片机嵌入式系统的结构如图 1-2 所示，通常包括三大部分：能实现嵌入式对象各种应用要求的单片机、全部系统的硬件电路和应用软件。

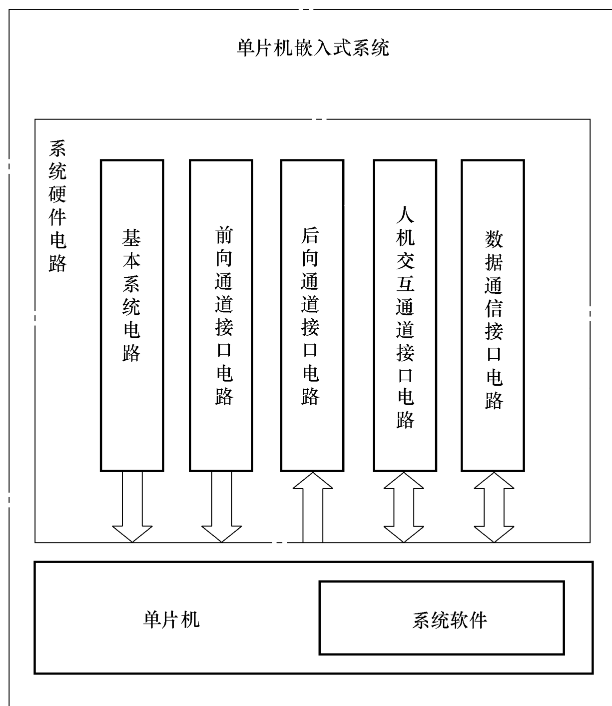


图 1-2 单片机嵌入式系统的结构

1. 单片机

单片机是单片机嵌入式系统的核心控制芯片，由它实现对控制对象的测控、系统运行管理控制和数据运算处理等功能。

2. 系统硬件电路

根据系统采用单片机的特性以及嵌入对象要实现的功能要求而配备的外围芯片、器件所构成的全部硬件电路。通常包括以下几部分：

1）基本系统电路。提供和满足单片机系统运行所需要的时钟电路、复位电路、系统供电电路、驱动电路、扩展的存储器等。

2）前向通道接口电路。这是应用系统面向对象的输入接口，通常是各种物理量的测量传感器、变换器输入通道。根据现实世界物理量转换成电量输出信号的类型，如模拟电压电流、开关信号、数字脉冲信号等的不同，接口电路也不同。常见的有传感器、信号调理器、A/D 转换器、开关信号输入、频率测量接口等。

3）后向通道接口电路。这是应用系统面向对象的输出控制电路接口。根据应用对象伺服和控制要求，通常有 A/D 转换器、开关量输出、功率驱动接口、PWM 输出控制等。

4）人机交互通道接口电路。人机交互通道接口电路是满足应用系统人机交互需要的电路，有键盘、拨动开关、LED 发光二极管、数码管、LCD 液晶显示器、打印机等多种输入

输出接口电路。

5）数据通信接口电路。数据通信接口电路是满足远程数据通信或构成多机网络应用系统的接口。通常有 RS232、PSI、I^2C、CAN 总线、USB 总线等通信接口电路。

3. 系统应用软件

系统应用软件的核心就是下载到单片机中的系统运行程序。整个嵌入式系统全部硬件的相互协调工作、智能管理和控制都由系统运行程序决定。它可认为是单片机嵌入式系统核心的核心。一个系统应用软件设计得好坏，往往也决定了整个系统性能的好坏。

系统软件是根据系统功能要求设计的，一个嵌入式系统的运行程序实际上就是该系统的监控与管理程序。对于小型系统的应用程序，一般采用汇编语言编写。而对于中型和大型系统的应用程序，往往采用高级程序设计语言（如 C 语言、Basic 语言）来编写。

编写嵌入式系统应用程序与编写其他类型的软件程序（如基于 PC 的应用软件设计开发）有很大的不同，嵌入式系统应用程序更加面向硬件低层和控制，而且还要面对有限的资源（如有限的 RAM）。因为嵌入式系统的应用软件不仅要直接面对单片机以及与它连接的各种不同种类和设计的外围硬件电路编程，还要面对系统的具体应用和功能编程。整个运行程序常常是输入输出接口、存储器、外围芯片、中断处理等多项功能交织在一起。因此，除了硬件系统的设计，系统应用软件的设计也是嵌入式系统开发研制过程中重要和困难的任务。需要强调说明的是，针对单片机嵌入式系统的硬件设计和软件设计两者之间的关系是十分紧密，互相依赖和制约的。因此，通常要求嵌入式系统的开发人员既要具备扎实的硬件设计能力，同时也要具备相当优秀的软件程序设计能力。

1.2.2　单片机嵌入式系统的应用领域

以单片机为核心构成的单片机嵌入式系统已成为现代电子系统中最重要的组成部分。在现代的数字化世界中，单片机嵌入式系统已经大量地渗透到我们生活的各个领域，几乎很难找到哪个领域没有单片机的踪迹。导弹的导航装置、飞机上各种仪表的控制、计算机的网络通信与数据传输、工业自动化过程的实时控制和数据处理、生产流水线上的机器人、医院里先进的医疗器械和仪器、广泛使用的各种智能 IC 卡、小朋友的程控玩具和电子宠物都是典型的单片机嵌入式系统应用。

由于单片机芯片的微小体积、极低的成本和面向控制的设计，使得它作为智能控制的核心器件被广泛地用于嵌入到工业控制、智能仪器仪表、家用电器、电子通信产品等各个领域的电子设备和电子产品中，主要的应用领域有以下几个方面。

1）智能家用电器。俗称带“电脑”的家用电器，如电冰箱、空调、微波炉、电饭锅、电视机、洗衣机等。传统的家用电器中嵌入了单片机系统后使产品的性能特点都得到很大的改善，实现了运行智能化、温度的自动控制和调节、节约电能等。

2）智能机电一体化产品。单片机嵌入式系统与传统的机械产品相结合，使传统的机械产品结构简化，控制智能化，构成新一代的机电一体化产品。这些产品已在纺织、机械、化工、食品等工业生产中发挥出巨大的作用。

3）智能仪器仪表。用单片机嵌入式系统改造原有的测量、控制仪器和仪表，能促使仪器仪表向数字化、智能化、多功能化、综合化、柔性化发展。由单片机系统构成的智能仪器

仪表可以集测量、处理、控制功能与一体，赋予传统的仪器仪表以崭新的面貌。

4）测控系统。用单片机嵌入式系统可以构成各种工业控制系统、适应控制系统、数据采集系统等。例如，温室人工气候控制、汽车数据采集与自动控制系统。

1.3　AVR 单片机简介

1.3.1　ATMEL 公司的单片机简介

ATMEL 公司是世界上著名的生产高性能、低功耗、非易失性存储器和各种数字模拟 IC 芯片的半导体制造公司。在单片微控制器方面，ATMEL 公司有基于 8051 内核、基于 AVR 内核和基于 ARM 内核的三大系列单片机产品（确切地讲，最后一款应称为嵌入式微处理器）。ATMEL 公司在它的单片机产品中，融入了先进的 EEPROM（Electrically Erasable Programmable Read-Only Memory，电可擦可编程只读存储器）和 Flash ROM（Flash Read-Only Memory，闪速存储器）技术，使得该公司的单片机具备了优秀的品质，在结构、性能和功能等方面都有明显的优势。

ATMEL 公司把 8051 内核与其擅长的 Flash 存储器技术相结合，是国际上最早推出片内集成可重复擦写 1000 次以上 Flash 程序存储器、采用低功耗 CMOS 工艺的 8051 兼容单片机的生产商之一。市场上家喻户晓的 AT89C51、AT89C52、AT89C1051、AT89C2051 就是 ATMEL 公司生产的基于 8051 内核系列单片机中的典型产品（现在已升级换代为 AT89S××系列），采用 ISP（在线编程）技术。该系列单片机一直在我国的单片机市场上占有相当大的份额。

8051 结构的单片机采用 CISC（Complex Instruction Set Computer，复杂指令系统计算机）体系。由于 CISC 结构存在指令系统不等长、指令数多、CPU 利用效率低、执行速度慢等缺陷，已不能满足和适应设计中高档电子产品和嵌入式系统应用的需要。ATMEL 公司发挥其 Flash 存储器技术的特长，于 1997 年研发和推出了全新配置采用 RISC（Reduced Instruction Set Computer，精简指令集计算机）结构的新型单片机，简称 AVR 单片机（由 ATMEL 公司挪威设计中心的 A 先生与 V 先生利用 ATMEL 公司的 Flash 新技术共同研发出 RISC 结构的高速 8 位单片机）。

RISC 结构是 20 世纪 90 年代开发出来的一种综合了半导体集成技术和提高软件性能的新结构，是为了提高 CPU 运行的速度而设计的芯片体系。它的关键技术在于采用流水线操作（Pipelining）和等长指令体系结构，使一条指令可以在一个单独操作中完成，从而实现在一个时钟周期里完成一条或多条指令。同时 RISC 体系还采用了通用快速寄存器组的结构，大量使用寄存器之间的操作，简化了 CPU 中处理器、控制器和其他功能单元的设计。因此，RISC 的特点就是通过简化 CPU 的指令功能，使指令的平均执行时间减少，从而提高 CPU 的性能和速度。在使用相同的晶片技术和相同的运行时钟条件下，RISC 系统的运行速度是 CISC 的 2～4 倍。正由于 RISC 体系所具有的优势，使得它在高端系统得到了广泛的应用。例如，ARM 以及大多数 32 位的处理器都采用 RISC 体系结构。

ATMEL 公司的 AVR 单片机是 8 位单片机中第一个真正的 RISC 结构的单片机。它采用了大型快速存取寄存器组、快速的单周期指令系统以及单级流水线等先进技术，使得 AVR

单片机具有高达 1MIPS/MHz 的高速运行处理能力。

AVR 单片机采用流水线技术，在前一条指令执行时，就取出现行的指令，然后以一个周期执行指令，大大提高了 CPU 的运行速度。而在其他的 CISC 以及类似的 RISC 结构的单片机中，外部振荡器的时钟被分频降低到传统的内部指令执行周期，这种分频最大达 12 倍（8051）。

另外，传统的基于累加器的结构单片机（如 8051），需要大量的程序代码来完成和实现在累加器和存储器之间的数据传送。而在 AVR 单片机中，由于采用 32 个通用工作寄存器构成快速存取寄存器组，用 32 个通用工作寄存器代替了累加器，从而避免了在传统结构中累加器和存储器之间数据传送造成的瓶颈现象，进一步提高了指令的运行效率和速度。

随着电子产品更新换代的周期缩短以及不断向高端发展，为了加快产品进入市场的时间和简化系统的设计、开发、维护和支持，对于以单片机为核心所组成的高端嵌入式系统来说，用高级语言编程已成为一种标准设计方法。AVR 单片机采用 RISC 结构，其目的就是在于能够更好地采用高级语言（例如 C 语言、Basic 语言）来编写嵌入式系统的系统程序，从而能高效地开发出目标代码。

AVR 单片机采用低功率、非挥发的 CMOS 工艺制造，内部分别集成 Flash、EEPROM 和 SRAM（Static Random Access Memory，静态随机存取存储器）三种不同性能和用途的存储器。除了可以通过使用一般的编程器（并行高压方式）对 AVR 单片机的 Flash 程序存储器和 EEPROM 数据存储器进行编程外，大多数的 AVR 单片机还具有 ISP（在线可编程）的特点以及 IAP（在应用编程）的特点。这些优点为使用 AVR 单片机开发设计和生产产品提供了极大的方便。在产品的设计生产中，可以“先装配后编程”，从而缩短了研发周期、工艺流程，并且还可以节约购买开发仿真编程器的费用。同样，对于学习和使用 AVR 单片机的用户来说，也不必购买昂贵的开发仿真硬件设备，只需要具备一套好的 AVR 开发软件平台，就可以从事 AVR 单片机系统的学习、设计和开发工作了。

1.3.2　AVR 单片机的主要特点

AVR 单片机吸取了 PIC 及 8051 等单片机的优点，同时在内部结构上还作了一些重大改进，其主要的优点如下：

1）程序存储器为价格低廉、可擦写 1 万次以上、指令长度单元为 16 位（字）的 Flash ROM（即程序存储器宽度为 16 位，按 8 位字节计算时应乘 2）。而数据存储器为 8 位。因此，AVR 单片机还是属于 8 位单片机。

2）采用 CMOS 技术和 RISC 架构，实现高速（50ns）、低功耗（μA）、具有 SLEEP（休眠）功能。AVR 单片机的一条指令执行速度可达 50ns（20MHz），而耗电则在 1μA ~2.5mA 之间。AVR 单片机采用 Harvard 结构，以及一级流水线的预取指令功能，即对程序的读取和数据的操作使用不同的数据总线，因此，当执行某一指令时，下一指令被预先从程序存储器中取出，这使得指令可以在每一个时钟周期内被执行。

3）高度保密。可多次烧写的 Flash 具有多重密码保护锁定（LOCK）功能，因此可低价快速完成产品商品化，且可多次更改程序（产品升级），方便了系统调试，而且不必浪费 IC 或电路板，大大提高了产品质量及竞争力。

4）工业级产品。具有大电流 10～20mA（输出电流）或 40mA（吸电流）的特点，可直接驱动 LED、SSR 或继电器。有看门狗定时器（WDT）安全保护，可防止程序跑飞，提高产品的抗干扰能力。

5）超功能精简指令。具有 32 个通用工作寄存器（相当于 8051 中的 32 个累加器），克服了单一累加器数据处理造成的瓶颈现象。片内含有 128～4KB SRAM，可灵活使用指令运算，适合使用功能很强的 C 语言编程，易学、易写、易移植。

6）程序写入器件时，可以使用并行方式写入（用编程器写入），也可使用串行在线下载（ISP）、在应用下载（IAP）方法下载写入。也就是说，不必将单片机芯片从系统板上拆下拿到万用编程器上烧录，而可直接在电路板上进行程序的修改、烧录等操作，方便产品升级，尤其是对于使用 SMD（Surface Mounted Devices，表贴封装器件），更利于产品微型化。

7）通用数字 I/O 口的输入输出特性与 PIC 单片机的 HI/LOW 输出及三态高阻抗 HI-Z 输入类似，同时可设定类似于 8051 结构内部有上拉电阻的输入端功能，便于为各种应用特性所需（多功能 I/O 口），AVR 单片机的 I/O 口是真正的 I/O 口，能正确地反映 I/O 口的输入/输出的真实情况。

8）单片机内集成有模拟比较器，可组成廉价的 A/D 转换器。

9）像 8051 一样，有多个固定中断向量入口地址，可快速地响应中断，而不是像 PIC 一样所有中断都在同一向量地址，需要通过程序判别后才可响应，这会浪费且失去最佳控制时机。

10）与 PIC 单片机一样，带有可设置的启动复位延时计数器。AVR 单片机内部有电源上电启动计数器，当系统 RESET 复位上电后，利用内部的 RC 看门狗定时器，可延迟 MCU 正式开始读取指令执行程序的时间。这种延时启动的特性，可使 MCU 在系统电源、外部电路达到稳定后再正式开始执行程序，提高了系统工作的可靠性，同时也可节省外加的复位延时电路。

11）具有多种不同方式的休眠省电功能和低功耗的工作方式。

12）许多 AVR 单片机具有内部的 RC 振荡器，提供 1MHz/2MHz/4MHz/8MHz 的工作时钟，使该类单片机无需外加时钟电路元器件即可工作，非常简单和方便。

13）有多个带预分频器的 8 位和 16 位功能强大的计数器/定时器（C/T），除了实现普通的定时和计数功能外，还具有输入捕获、产生 PWM 输出等更多的功能。

14）性能优良的串行同步/异步通信 USART 口，不占用定时器，可实现高速同步/异步通信。

15）MEGA 8515 及 MEGA 128 等芯片具有可并行扩展的外部接口，扩展能力达 64KB。

16）工作电压范围宽（2.7～6.0）V，具有系统电源低电压检测功能，电源抗干扰性能强。

17）有多通道的 10 位 A/D 接口及实时时钟（RTC）。许多 AVR 芯片内部集成了 8 路 10 位 A/D 接口，如 MEGA 8、MEGA 16、MEGA 8535 等。

AVR 单片机还在片内集成了可擦写 10 万次的 EEPROM 数据存储器，等于又增加了一个芯片，可用于保存系统的设定参数、固定表格和掉电后的数据保存，既方便了使用，减小了系统的空间，又大大提高了系统的保密性。

1.3.3　AVR 单片机最小系统

学过 MCS-51 系列单片机的读者应该知道，要使单片机工作则必须保证单片机自身最小系统电路的完整。以大家熟知的 MCS-51 系列单片机为例，它的硬件最小系统包括：电源、晶振、复位电路，有了这些硬件电路支持，单片机才有可能正常运行，甚至 MCS-51 系列单片机要使用 P0 口时还要外接上拉电阻。相比之下，AVR 系列单片机比 MCS-51 则要简单得多，因为 AVR 系列单片机有完善的内部模块电路及熔丝。AVR 系列单片机内部都有复位电路、RC 振荡电路，因此在一般使用中最小系统基本不用外围电路，只要给它供电就能独立工作。AVR 单片机内部 RC 振荡器可以给系统提供时钟信号，且这个振荡频率可以通过修改熔丝位来改变频率高低，一般 AVR 系列单片机在出厂时把熔丝位统一设成内部 1MHz 的 RC 振荡。由此可知，在实验中 AVR 单片机的硬件最小系统基本不用再加其他硬件电路。

第 2 章　AVR 单片机的基本结构

2.1　单片机的基本组成

2.1.1　单片机的基本组成结构

学过 MCS-51 单片机的读者都知道单片机的基本组成，即单片机是由中央处理器（CPU 中的运算器和控制器）、只读存储器（通常表示为 ROM）、读写存储器（又称随机存储器，通常表示为 RAM）、输入/输出口（又分为并行口和串行口，表示为 I/O 口）等组成。实际上单片机里面还有一个时钟电路，使单片机在进行运算和控制时，都能有节奏地进行。另外，还有所谓的“中断系统”，这个系统有“传达室”的作用，当单片机控制对象的参数到达某个需要加以干预的状态时，就可经此“传达室”通报给 CPU，使 CPU 根据外部事态的轻重缓急来采取适当的应付措施。

那么，这么多组成模块是如何共同工作的呢？如图 2-1 所示，单片机内部有一条将它们连接起来的“纽带”，即所谓的“内部总线”。此总线有如大城市的“干道”，而 CPU、ROM、RAM、I/O 口、中断系统等就分布在此总线的两旁，并与它连通。从而，一切指令、数据都可经内部总线传送，有如大城市内各种物品的传送都经过干道进行。

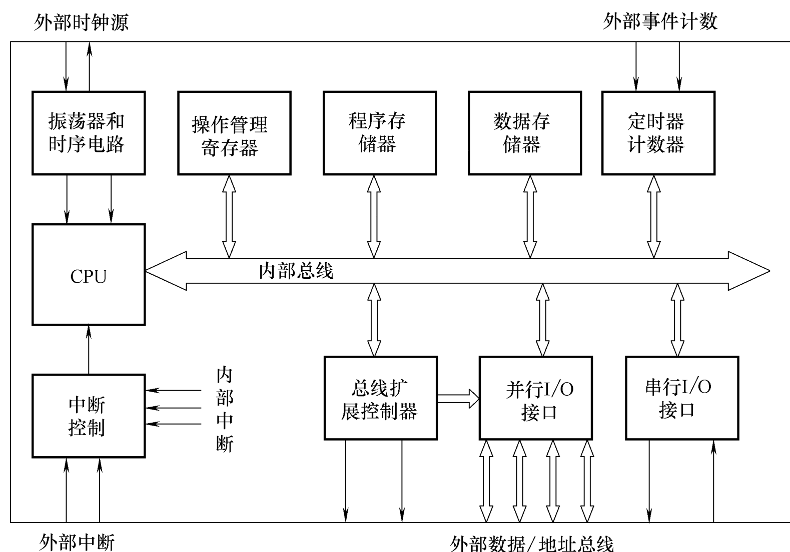


图 2-1　常见单片机的基本组成结构

2.1.2　单片机的基本单元与作用

典型的单片机包括了 MCU 单元、片内存储器、程序存储器、数据存储器、I/O 口及寄存器。各组成部分功能如下：

1. MCU 单元

MCU（Microcontroller Unit）单元部分包括了 CPU、时钟系统、复位、总线控制逻辑等电路。CPU 是按照面向测控对象、嵌入式应用的要求设计的，其功能有进行算术、逻辑、比较等运算和操作，并将结果和状态信息与存储器以及状态寄存器进行交换（读/写）。时钟和复位电路实现上电复位、信号控制复位，产生片内各种时钟及功耗管理等。总线控制电路则产生各类控制逻辑信号，满足 MCU 对内部和外部总线的控制。其中，内部总线用以实现片内各单元电路的协调操作和数据传输，而外部总线控制用于单片机外围扩展的操作管理。

2. 片内存储器

单片机的存储器一般分为程序存储器和数据存储器，它们往往构成相互独立的两个存储空间，分别寻址，互不干扰。在这一点上，与通用计算机系统的结构是不同的。通用计算机系统通常采用冯·诺依曼结构，在这种结构体系中采用了单一的数据总线用于指令和数据的存取，因此数据和指令是存放在同一个存储空间中的，CPU 使用同一条数据总线与数据和程序进行交换，如在计算机原理课程中介绍的 8086/8088 微处理器。而单片机的内部结构通常使用哈佛体系结构，在这种体系中采用分开的指令和数据总线，以及分开的指令和数据地址空间。单片机采用哈佛双（多）总线结构的优点是，指令和数据空间完全分开，分别通过专用的总线与 CPU 交换，可以实现对程序和数据的同时访问，提高了 CPU 的执行速度和数据的吞吐率。

早期的单片机，如典型的 8031 单片机，在片内只集成少量的数据存储器 RAM（128/256B），没有程序存储器。因此，程序存储器和大容量的数据存储器需要进行片外的扩展，增加外围的存储芯片和电路，这给构成嵌入式系统带来了麻烦。后期的单片机则在片内集成了相当数量的程序存储器，如与 8031 兼容的 AT89S51、AT89S52 在片内集成了 4K/8K 的 Flash 程序存储器。而现在新型的单片机，则在片内集成了更多数量和更多类型的存储器。如 AVR 系列的 ATmega16 在片内就集成了 16KB 的 Flash 程序存储器，1KB 的 RAM 数据存储器，以及 512B 的 EEPROM 数据存储器，这就大大方便了应用。

3. 程序存储器

程序存储器用于存放嵌入式系统的应用程序。由于单片机嵌入式系统的应用程序在开发调试完成后不需要经常改变，因此单片机的程序存储器多采用只读存储器（ROM），用于永久性存储系统的应用程序。为适应不同的产品、用户和场合的需要，单片机的程序存储器有以下几种不同形式：

1）ROMLess 型。这种形式的单片机片内没有集成程序存储器，使用时必须在单片机外部扩展一定容量的 EPROM 器件。因此，使用这种类型的单片机就必须使用并行扩展总线，增加芯片，增加硬件设计的工作量。

2）EPROM 型。单片机片内集成了一定数量的 EPROM 存储器用于存放系统的应用程序。这类单片机芯片的上部开有透明窗口，可通过约 15min 的紫外线照射来擦除存储器中的程序，再使用专用的写入装置写入程序代码和数据，写入次数一般为几十次。

3）MaskROM 型。使用这种类型的单片机时，用户要将调试好的应用程序代码交给单片机的生产厂商，生产厂商在单片机芯片制造过程的掩膜工艺阶段将程序代码掩膜到程序存储器中。这种单片机便成为永久性专用的芯片，系统程序无法改动，适合于大批量产品的生产。

4）OTPROM 型。这种类型的单片机与 MaskROM 型的单片机有相似的特点。生产厂商提供新的单片机芯片中程序存储器可由用户使用专用的写入装置一次性编程写入程序代码，写入后也无法改动了。这种类型的单片机也是适用于大批量产品的生产。

5）FlashROM 型。这是一种可供用户多次擦除和写入程序代码的单片机。它的程序存储器采用快闪存储器（FlashMemory），现在可实现大于 1 万次的写入操作。

内部集成 FlashROM 型单片机的出现，以及随着 Flash 存储器价格的下降，使得使用 FlashROM 的单片机正在逐步淘汰使用其他类型程序存储器的单片机。由于 FlashROM 可多次擦除（电擦除）和写入的特性，加上新型的单片机又采用了在线下载 ISP 技术（In System Program，即无需将芯片从系统板上取下，直接在线将新的程序代码写入单片机的程序存储器中），不仅为用户在嵌入式系统的设计、开发和调试带来了极大的方便，而且也适用于大批量产品的生产，并为产品的更新换代提供了更广阔的空间。

4. 数据存储器

单片机在片内集成的数据存储器一般有两类：随机存储器（RAM）和电可擦除存储器（EEPROM）。

1）随机存储器（RAM）。在单片机中，随机存储器（RAM）是用来存储系统程序在运行期间的工作变量和临时数据的。一般在单片机内部集成一定容量（32～512B 或更多）的 RAM。这些小容量的数据存储器以高速 RAM 的形式集成在单片机芯片内部，作为临时的工作存储器使用，可以提高单片机的运行速度。

在单片机中，常把内部寄存器（如工作寄存器、I/O 寄存器等）在逻辑上也划分在 RAM 空间中，这样既可以使用专用的寄存器指令对寄存器进行操作，也可将寄存器当做 RAM 使用，为程序设计提供了方便和灵活性。

对一些需要使用大容量数据存储器的系统，就需要在外部扩展数据存储器。这时，单片机就必须具备并行扩展总线的功能，同时外围也要增加 RAM 芯片和相应的地址锁存、地址译码等电路。这不仅增加了硬件设计的工作量，产品的成本，同时降低了系统的可靠性。

目前，许多新型单片机片内集成的 RAM 容量越来越大。片内集成的 RAM 容量增加，不仅减少了在片外扩展 RAM 的必要性，提高了系统的可靠性，而且更重要的是，使得单片嵌入式系统的软件设计思想和方法有了许多改变和发展，给编写系统程序带来很大的方便，更加有利于结构化、模块化的程序设计。

2）电可擦除存储器（EEPROM）。一些新型的单片机，在芯片中还集成了电可擦除存储器（EEPROM）的数据存储器。这类数据存储器用于存放一些永久或比较固定的系统参数，如放大倍率、电话号码、时间常数等。EEPROM 的寿命大于 10 万次，具有掉电后不丢失数据的特点，并且通过系统程序可以随时修改，这些特性都给用户设计开发产品带来极大的方便和想象空间。

5. 输入/输出（I/O）口

为了满足嵌入式系统“面向控制”的实际应用需要，单片机提供了数量众多、功能强、使用灵活的输入/输出口，简称 I/O 口。端口的类型可分为以下几种：

1）并行总线输入/输出口（并行 I/O 口）。用于外部扩展和扩充并行存储器芯片或并行 I/O 芯片等使用，包括数据总线、地址总线和读写控制信号等。

2）通用数字 I/O 口。用于外部电路逻辑信号的输入和输出控制。

3）片内功能单元的输入/输出口。如定时器/计数器的计数脉冲输入，外部中断源信号

的输入等。

4）串行 I/O 通信口。用于系统之间或与采用专用串行协议的外围芯片之间的连接和交换数据。如 UART 串行接口（RS232），I^2C 串行接口，SPI 串行接口，USB 串行口等。

5）其他专用接口。一些新型的单片机还在片内集成了某些专用功能的模拟或数字 I/O 口，如 A/D 输入、D/A 输出接口，模拟比较输入端口，脉宽调制（PWM）输出端口等。有的单片机还将 LCD 液晶显示器的接口也集成到单片机芯片中了。

为了减少芯片引脚的数量，又能提供更多性能的 I/O 口给用户使用，大多数的单片机都采用了 I/O 口复用技术，即某一端口，它既可作为一般通用的数字 I/O 口使用，也可作为某个特殊功能的端口使用，用户可根据系统的实际需要来定义使用。这样就为设计开发提供了方便，大大拓宽了单片机的应用范围。

6. 操作管理寄存器

操作管理寄存器也是单片机芯片中的重要组成部分之一。它的功能是管理、协调、控制和操作单片机芯片中的各功能单元的使用和运行。这类寄存器的种类有状态寄存器、控制寄存器、方式寄存器、数据寄存器等。各种寄存器的定义、功能、状态、相互之间的关系和应用相对比较复杂，而且往往与相应的功能单元的使用紧密相关。因此，用户应非常熟悉各个寄存器的作用以及如何与不同的功能单元的配合使用，这样才能通过程序指令对其编程操作，以实现对单片机芯片中各种功能的正确使用，充分发挥单片机的所有特点和性能，设计和开发出高性能、低成本的电子产品。可以这样讲，当你对某个单片机芯片中各个操作管理寄存器的作用、功能、定义非常透彻地掌握了，那么你已经完全精通和能够熟练使用该单片机了。

2.2　ATmega16 单片机的组成

ATMEL 公司的 AVR 单片机是一种基于增强 RISC 结构的、低功耗、CMOS 技术、8 位微控制器（Enhanced RISC Microcontroller），目前有 Tiny、Mega 两个系列 50 多种型号。虽然功能和外部的引脚各有不同，但它们内核的基本结构是一样的，指令系统相容。本书将会以性能适中的 ATmega16 单片机作为主要对象，介绍 AVR 单片机的原理与应用。在实际工程中读者可以根据实际功能要求选择其他型号单片机作为控制核心。

2.2.1　AVR 单片机的内核结构

虽然 AVR 单片机系列有几十种型号，但是它们有着相同的内核结构，指令兼容。图 2-2 所示为典型的 AVR 单片机的内核结构图。

为了提高 MCU 并行处理的运行效率，AVR 单片机采用了哈佛结构，具有独立的数据和程序总线。程序存储器里的指令通过一级流水线运行。CPU 在执行一条指令的同时读取下一条指令（ 在本文称为预取）。这个概念实现了指令的单时钟周期运行。

在 AVR 单片机的内核中，由 32 个访问操作只需要一个时钟周期的 8 位通用工作寄存器组成了“快速访问寄存器组”。快速访问寄存器访问时间为一个时钟周期，从而实现了单时钟周期的 ALU（Arithmetic Logic Unit，算术逻辑单元）操作。在典型的 ALU 操作中，两个位于寄存器文件中的操作数同时被访问，然后执行运算，结果再被送回到寄存器文件，整个过程仅需一个时钟周期。寄存器文件里有 6 个寄存器可以用作 3 个 16 位的间接寻址寄存器

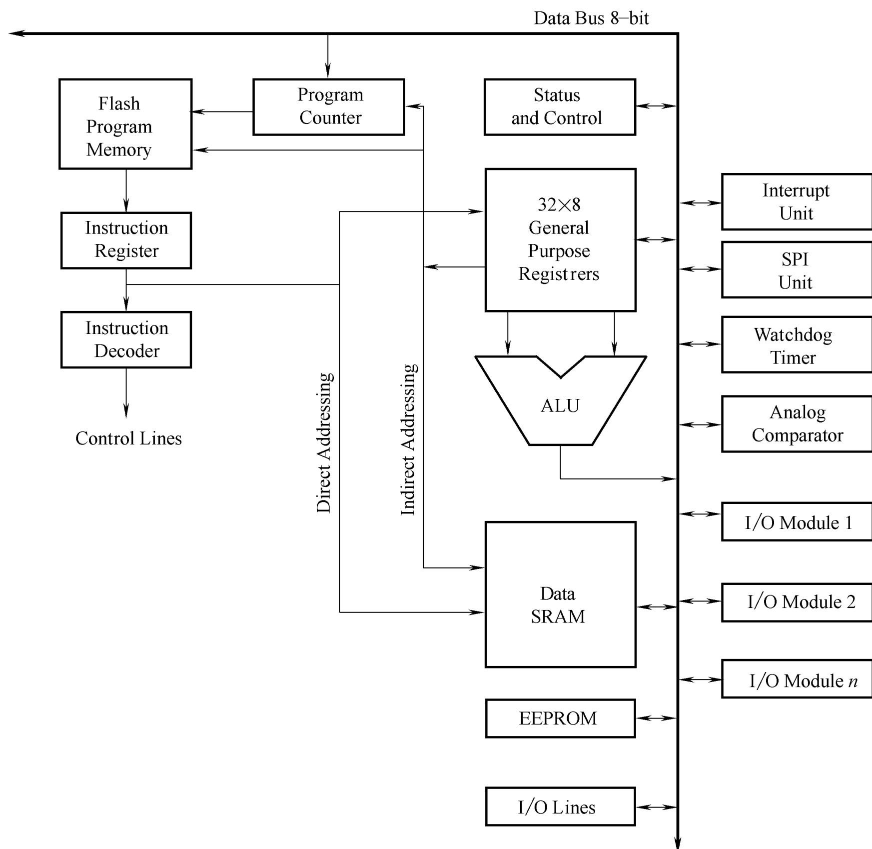


图 2-2　AVR 单片机的内核结构示意图

指针以寻址数据空间，实现高效的地址运算。这 3 个 16 位的间接寻址寄存器称为 X 寄存器、Y 寄存器和 Z 寄存器。其中 Z 寄存器还能作为间接寻址程序存储器空间的地址寄存器，用于在 Flash 程序存储器空间进行查表等操作。

ALU 支持寄存器之间以及寄存器和常数之间的算术和逻辑运算。ALU 也可以执行单寄存器操作。运算完成之后状态寄存器的内容得到更新以反映操作结果。程序流程通过有/无条件的跳转指令和调用指令来控制，从而直接寻址整个地址空间。大多数指令长度为 16 位，亦即每个程序存储器地址都包含一条 16 位或 32 位的指令。

单片机的 Flash 程序存储器空间可以分成两段：引导程序段（Boot Program Section）和应用程序段（Application Program Section）。两个段的读写保护可以分别通过设置对应的锁定位（Lock Bits）来实现。在引导程序段内驻留的引导程序中，可以使用 SPM（Scratch Pad Memory，便笺式存储器）指令，实现对应用程序段的写操作（实现在应用编程（IAP）功能，使系统能够自己更新系统程序）。在 AVR 单片机中，所有的存储器空间都是线性的。数据存储器（SRAM）可以通过 5 种不同的寻址方式进行访问。

AVR 单片机有一个灵活的中断模块。在响应中断服务和子程序调用过程时，程序计数器（PC）中的返回地址将被存储于堆栈之中。堆栈空间将占用数据存储器（SRAM）中一段连续的地址。因此，堆栈空间的大小仅受到系统总的数据存储器（SRAM）的大小以及系统程序对 SRAM 的使用量的限制。用户程序应在系统上电复位后，对一个 16 位的堆栈指针

寄存器（SP）进行初始化设置（或在子程序和中断程序被执行之前）。

AVR 单片机的中断控制由 I/O 寄存器空间的中断控制寄存器和状态寄存器中的全局中断允许位组成。每个中断都分别对应一个中断向量（中断入口地址）。所有的中断向量构成了中断向量表，该中断向量表位于 Flash 程序存储器空间的最前面。中断的中断向量地址越小，其中断的优先级越高。

I/O 空间为连续的 64 个 I/O 寄存器空间，它们分别对应 MCU 各个外围功能的控制和数据寄存器地址，如控制寄存器、定时器/计数器、A/D 转换器及其他的 I/O 功能等。I/O 寄存器空间可使用 I/O 寄存器访问指令直接访问，也可将其映射为通用工作寄存器组后的数据存储器空间，使用数据存储器访问指令进行操作。I/O 寄存器空间在数据存储器空间的映射地址为 $020 ~ $05F。

因为 AVR 单片机的性能非常强大，所以它的内部结构相对 MCS-51 结构的单片机要复杂。对于没有学过 MCS-51 单片机的读者来说可能会感觉很难理解，但通过后面章节的介绍及应用实例，会对学习 AVR 单片机有很大的帮助。

2.2.2　ATmega16 的特点

AVR 系列单片机中比较典型的芯片是 ATmega16。这款芯片具备了 AVR 系列单片机的主要的特点和功能，不仅适合应用于产品设计，同时也方便初学者入门。其主要特点有：

（1）采用先进 RISC 结构的 AVR 内核

131 条机器指令，且大多数指令的执行时间为单个系统时钟周期；32 个 8 位通用工作寄存器；工作在 16MHz 时具有 16MIPS 的性能；只需要配备 2 个时钟周期的硬件乘法器。

（2）片内含有较大容量的非易失性的程序和数据存储器

16KB 在线可编程（ISP）Flash 程序存储器（擦除次数大于 1 万次），采用 Boot Load 技术支持 IAP 功能；1KB 的片内 SRAM 数据存储器，可实现 3 级锁定的程序加密；512B 片内在线可编程 EEPROM 数据存储器（寿命大于 10 万次）。

（3）片内含 JTAG 接口

支持符合 JTAG（Joint Test Action Group，联合测试行为组织）标准的边界扫描功能用于芯片检测；支持扩展的片内在线调试功能；可通过 JTAG 接口对片内的 Flash、EEPROM、配置熔丝位和锁定加密位实施下载编程。

（4）外围接口

2 个分别带有独立、可设置预分频器的 8 位定时器/计数器；1 个带有可设置预分频器、具有比较、捕捉功能的 16 位定时器/计数器；片内含独立振荡器的实时时钟芯片（RTC）；4 路 PWM 通道；8 路 10 位 A/D 转换器；面向字节的两线通信接口（Two-wire Serial Interface，TWI）（兼容 I^2C 硬件接口）；1 个可编程的增强型全双工、支持同步/异步通信的串行接口 USART；1 个可工作于主机/从机模式的 SPI 串行接口（支持 ISP 程序下载）；片内模拟比较器；内含可编程的、具有独立片内振荡器的看门狗定时器（WDT）。

（5）其他特点

片内含上电复位电路以及可编程的掉电检测复位电路（BOD）；片内含有 1MHz/2MHz/4MHz/8MHz、经过标定的、可校正的 RC 振荡器，可作为系统时钟使用；多达 21 个各种类型的内外部中断源；有 6 种休眠模式支持省电方式工作。

（6）宽电压、高速度、低功耗

工作电压范围宽：ATmega16L 2.7 ~5.5V，ATmega16 4.5 ~5.5V；

运行速度：ATmega16L 0 ~8MHz，ATmega16 0 ~16MHz；低功耗：ATmega16L 工作在 1MHz、3V、25℃时的典型功耗：正常工作模式 1.1mA，空闲工作模式 0.35mA，掉电工作模式 <1μA。

（7）芯片引脚和封装形式

ATmega16 共有 32 个可编程的 I/O 口（脚），芯片封装形式有 40 引脚的 PDIP、44 引脚的 TQFP 和 44 引脚的 MLF 封装。

2.2.3　ATmega16 的外部引脚与封装

ATmega16 单片机有 3 种形式的封装：40 脚双列直插 PDIP、44 脚方形的 TQFP 和 MLF 形式（贴片形式）。其外部引脚封装如图 2-3 所示。

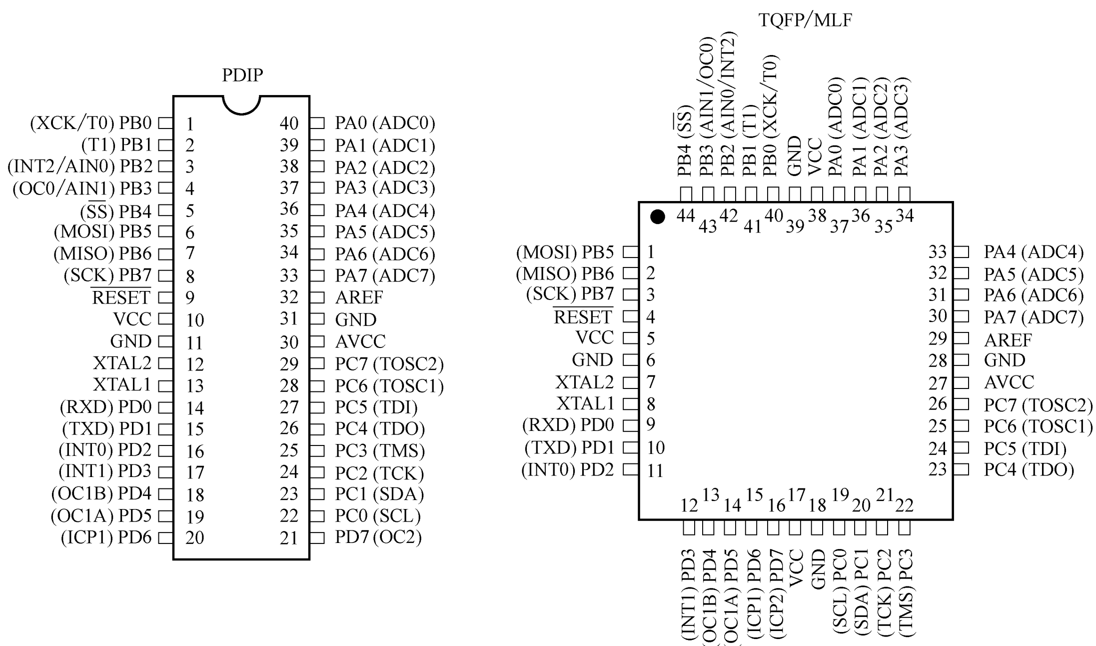


图 2-3　ATmega16 外部引脚与封装示意图

其中，各个引脚的功能如下：

（1）电源、系统晶振、芯片复位引脚

VCC：芯片供电（片内数字电路电源）输入引脚，使用时连接到电源正极。

AVCC：为端口 A 和片内 A/D 转换器模拟电路电源输入引脚。不使用 A/D 转换器时，直接连接到电源正极；使用 A/D 转换器时，应通过一个低通电源滤波器与 VCC 连接。

AREF：使用 A/D 转换器时，可作为外部 A/D 转换器参考源的输入引脚。

GND：芯片接地引脚，使用时接地。

XTAL2：片内反相振荡放大器的输出端。

XTAL1：片内反相振荡放大器和内部时钟操作电路的输入端。

$\overline{\text{RESET}}$：芯片复位输入引脚。在该引脚上施加（拉低）一个最小脉冲宽度为 1.5μs 的低电平，将引起芯片的硬件复位（外部复位）。

（2）32 根 I/O 引脚，分成 PA、PB、PC 和 PD 4 个 8 位端口，它们全部是可编程控制的双（多）功能复用的 I/O 引脚（口）

4 个端口的第一功能是通用的双向数字输入/输出（I/O）口，其中每一位都可以由指令设置为独立的输入口或输出口。当 I/O 设置为输入时，引脚内部还配置有上拉电阻，这个内部的上拉电阻可通过编程设置为上拉有效或上拉无效。

如果 AVR 单片机的 I/O 口设置为输出方式工作，当其输出高电平时，能够输出 20mA 的电流，而当其输出低电平时，可以吸收 40mA 的电流。因此，AVR 单片机的 I/O 口驱动能力非常强，能够直接驱动发光二极管（LED）、数码管等。而早期单片机 I/O 口的驱动能力只有 5mA，驱动 LED 时，还需要增加外部的驱动电路和器件。

芯片复位后，所有 I/O 口的默认状态为输入方式，上拉电阻无效，即 I/O 为输入高阻的三态状态。

以上可知，一片小小的 Mega16 芯片却有非常强大的功能。下面就从 Mega16 的内部结构出发，逐步地介绍它的工作原理和使用方法。

2.3　ATmega16 单片机的内部结构

图 2-4 所示为 ATmega16 的结构框图。它是在 AVR 单片机内核（见图 2-3）的基础上，具体化的一个实例。从图中可以看出，ATmega16 内部的主要构成部分有：

1）CPU 部分。包括：运算逻辑单元（ALU）、32 个 8 位快速访问通用寄存器组（寄存器文件）、程序计数器（PC）、指令寄存器、指令译码器。

2）程序存储器 Flash。

3）数据存储器 RAM 和 EEPROM。

4）各种功能的外围接口、I/O，以及与它们相关的数据、控制、状态寄存器等。

2.3.1　中央处理器

CPU 是单片机的核心部分，它由运算逻辑单元（ALU）、程序计数器（PC）、指令寄存器、指令译码器等部件组成。

1. 运算逻辑单元（ALU）

运算逻辑单元（ALU）的功能是进行算术运算和逻辑运算，可对半字节（4 位）、单字节等数据进行操作，如能完成加、减、自动加 1、自动减 1、比较等算术运算和与、或、异或、求补、循环移位等逻辑操作。操作结果的状态，如产生进位、结果为零等状态信息将影响到状态寄存器（SREG）相应的标志位。

运算逻辑单元（ALU）还包含一个布尔处理器，用来处理位操作。它可执行置位、清零、取反等操作。

ATmega16 的 ALU 还能实现无符号数、有符号数以及浮点数的硬件乘法操作。一次硬件乘法操作的时间为 2 个时钟周期。

2. 程序计数器（PC）、指令寄存器和指令译码器

程序计数器（PC）用来存放下一条需要执行指令在程序存储器空间的地址（指向 Flash 空间）。取出的指令存放在指令寄存器中，然后送入指令译码器产生各种控制信号，控制 CPU 的运行（执行指令）。

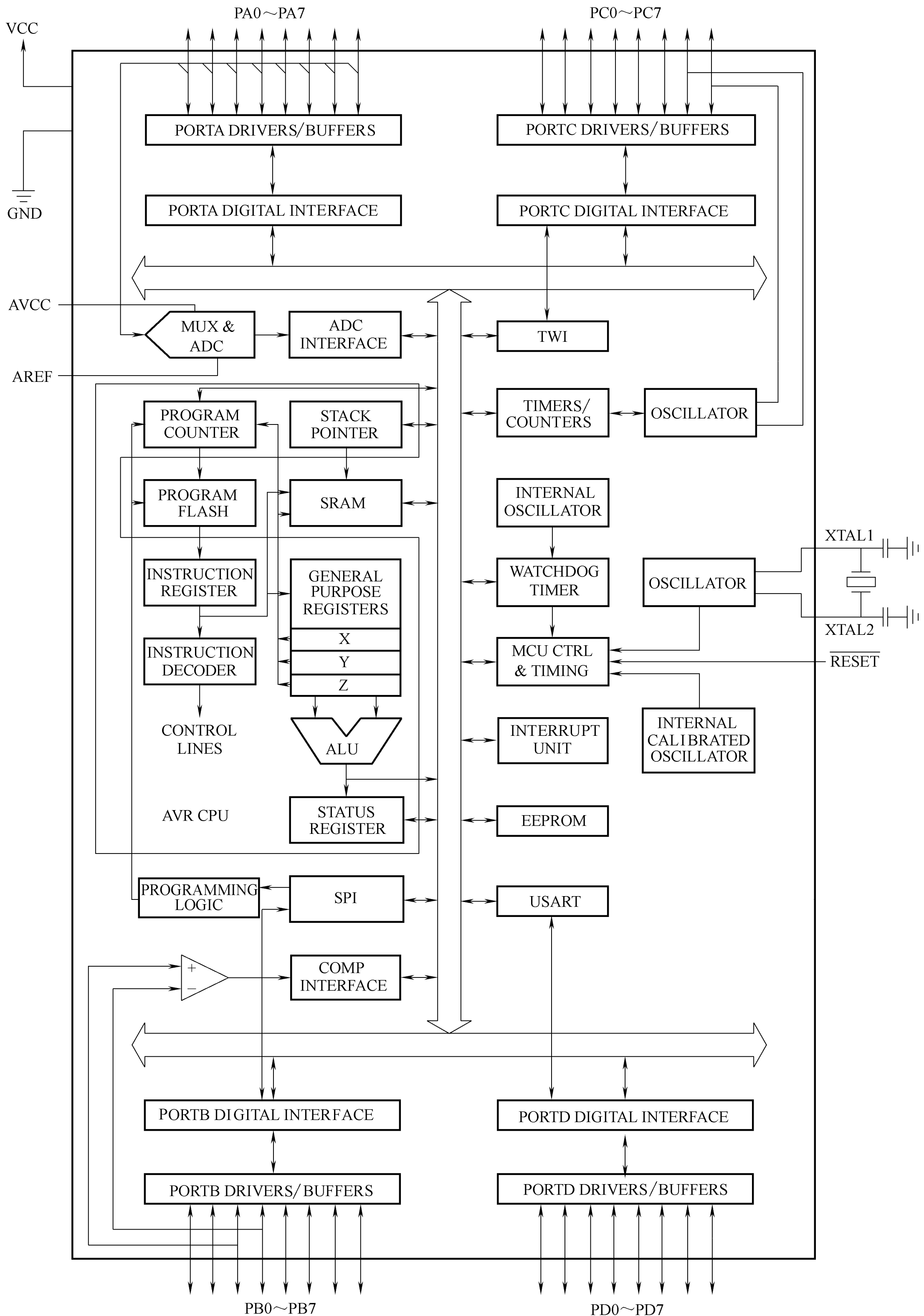


图 2-4　ATmega16 的结构框图

AVR 单片机一条指令的长度大多数为 16 位，还有少部分为 32 位，因此 AVR 单片机的程序存储器结构实际上是以字（16 位）为一个存储单元的。ATmega16 的程序计数器为 13 位，正好满足了对片内 8KB（即 16KB）的 Flash 程序存储器空间直接寻址的需要，因此就不能（不支持）在外部扩展更多的程序存储器。

AVR 单片机 CPU 在译码执行一条指令的同时，就将 PC 指定的 Flash 单元中的指令取出，放入指令寄存器中，构成了一级流水线运行方式。AVR 单片机采用一级流水线技术，在当前指令执行的时候，就取出下一条将要执行的指令，加上大多数 AVR 指令的长度是 1W，就使得 AVR 单片机 CPU 实现了一个时钟周期执行一条指令。采用这种结构，减少了取指令的次数，大大提高了 CPU 的运行速度，同时也提高了取指令操作的（系统的）可靠性。而在其他的 CISC 以及类似的 RISC 结构的单片机中，外部振荡器的时钟被分频降低到传统的内部指令执行周期，这种分频最大达 12 倍（例如，标准 8031 结构的单片机）。

3. 通用工作寄存器组

在 AVR 单片机中，由命名为 R0 ~ R31 的 32 个 8 位通用工作寄存器构成一个“通用快速工作寄存器组”，图 2-5 所示为通用快速工作寄存器组的结构图。

AVR 单片机 CPU 中的 ALU 与这 32 个通用工作寄存器组直接相连，为了使 ALU 能够高效和灵活地对寄存器组进行访问操作，通用寄存器组提供和支持 ALU 使用 4 种不同的数据输入/输出的操作方式：

1）提供一个 8 位源操作数，并保存的一个 8 位结果；

2）提供两个 8 位源操作数，并保存的一个 8 位结果；

3）提供两个 8 位源操作数，并保存的一个 16 位结果；

4）提供一个 16 位源操作数，并保存的一个 16 位结果。

因此，AVR 单片机大多数操作工作寄存器组的指令都可以直接访问所有的寄存器，而且多数这样的指令的执行时间是一个时钟周期。例如，从寄存器组中取出两个操作数，对操作数实施处理，处理结果回写到目的寄存器中。这 3 个过程是在一个时钟周期内完成的，构成一个完整的 ALU 指令操作。

在传统的基于累加器结构的单片机中（如 8051），则需要大量的程序代码来完成和实现在累加器和存储器之间的数据传送。如上面所介绍的操作过程就需要 3 条指令来实现：第 1 条完成从寄存器中取出源操作数；第 2 条完成对操作数实施处理；第 3 条将处理结果回写。这样就构成了累加器和存储器之间数据传送的瓶颈，影响了指令运行效率。

而在 AVR 单片机中，由于采用了 32 个通用工作寄存器构成快速存取寄存器组，相当于用 32 个通用工作寄存器代替了累加器，从而避免了在传统结构中的那种由于累加器和存储器之间频繁的数据传送交换而形成的瓶颈现象，又进一步提高了指令的运行效率和速度。

在 AVR 单片机中，通用寄存器组与片内的数据存储器（SRAM）处在相同的空间，32 个通用寄存器被直接映射到用于数据空间的前 32 个地址，如图 2-5 所示。虽然寄存器组的物理结构与 SRAM 不同，但是这种内存空间的组织方式为访问工作寄存器提供了极大的灵活性，如可以利用地址指针寄存器 X、Y 或 Z 实现对通用寄存器组的间接寻址操作。

2. 3. 2　系统时钟部件

1. 系统时钟

ATmega16 的片内含有 4 种频率（1MHz/2MHz/4MHz/8MHz）的 RC 振荡源，可直接作

为系统的工作时钟使用。同时片内还设有一个由反向放大器所构成的 OSC（Oscillator）振荡电路，外围引脚 XTAL1 和 XTAL2 分别为 OSC 振荡电路的输入端和输出端，用于外接石英晶体等，构成高精度的或其他标称频率的系统时钟系统。

系统时钟为控制器提供时钟脉冲，是控制器的“心脏”。系统时钟的频率是单片机的重要性能指标之一。系统时钟频率越高，单片机的执行节拍就越快，处理速度也越快。ATmega16 最高的工作频率为 16MHz（16MIPS），在 8 位单片机中算是佼佼者。但并不是系统时钟频率越快就越好，因为当时钟频率越高时，其耗电量也越大，也容易受到干扰（或干扰别的设备）。因此，在具体设计时，应根据实际产品的需要，尽量采用较低的系统时钟频率，这样不仅能降低功耗，同时也提高系统的可靠性和稳定性。

寄存器名	RAM空间地址	
R0	$0000	
R1	$0001	
R2	$0002	
⋮	⋮	
R14	$000E	
R15	$000F	
R16	$0010	
⋮	⋮	
R26	$001A	X寄存器低位字节
R27	$001B	X寄存器高位字节
R28	$001C	Y寄存器低位字节
R29	$001D	Y寄存器高位字节
R30	$001E	Z寄存器低位字节
R31	$001F	Z寄存器高位字节

图 2-5　通用工作寄存器组在 RAM 空间的地址分配图

为 ATmega16 提供系统时钟源时，有 3 种主要的选择：①直接使用片内 1MHz/2MHz/4MHz/8MHz 的 RC 振荡源；②在引脚 XTAL1 和 XTAL2 上外接由石英晶体和电容组成的谐振回路，配合片内的 OSC 振荡电路构成的振荡源；③直接使用外部时钟源输出的脉冲信号。方式 2 和方式 3 的电路连接如图 2-6 所示。

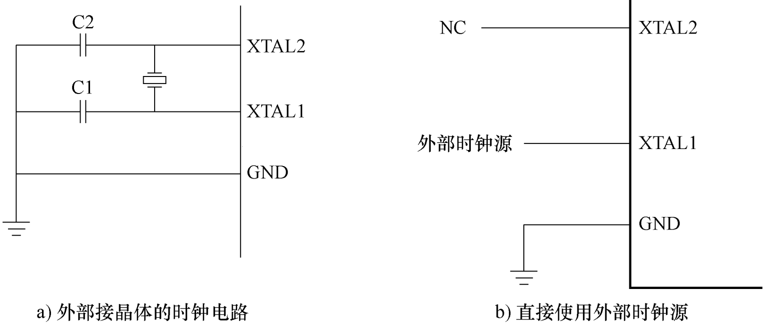


图 2-6　系统时钟源的电路连接

图 2-6a 是比较常用的方法，由于采用了外接石英晶体作为振荡的谐振回路，因此可以提供比较灵活的频率（由使用晶体的谐振频率决定）和稳定精确的振荡。在 XTAL1 和 XTAL2 引脚上加上由石英晶体和电容组成的谐振回路，与内部振荡电路配合就能产生系统需要的时钟信号了。最常采用的晶体元件为一个石英晶体和两个电容组成谐振电路。晶体可在 0～16MHz 之间选择，电容值在 20～30pF 之间（最好与所选用的晶体相匹配）。

当对系统时钟电路的精度要求不高的话，可以使用方式 1，即使用片内可选择的 1MHz/2MHz/4MHz/8MHz 的 RC 振荡源作为系统时钟源，可以节省外接器件，此时 XTAL1 和 XTAL2 引脚悬空。

系统时钟电路产生振荡脉冲不经过分频将直接作为系统的主工作时钟，同时它还作为芯片内部的各种计数脉冲，以及各种串口定时时钟等使用（可由程序设定分频比例）。

值得注意的是，AVR 单片机有一组专用的、与芯片功能、特性、参数配置相关的可编程熔丝位。其中有几个专门的熔丝位（CKSEL3..0）用于配置芯片所要使用的系统时钟源的类型。

新芯片的默认配置设定为使用内部 1MHz 的 RC 振荡源作为系统的时钟源。因此，当第一次使用前，必须先正确地配置熔丝位，使其与使用的系统时钟源类型相匹配。另外，在配置其他熔丝位时，或进行程序下载时，千万不要对 CKSEL3..0 这几个熔丝位误操作，否则会组成芯片表面现象上的“坏死”，因为没有系统时钟源，芯片不会工作的。

关于 ATmega16 重要熔丝位的配置、使用方式将在下文详细介绍。

2. 内部看门狗定时器

在 AVR 单片机片内还集成了一个 1MHz 独立的时钟电路，它仅供片内的看门狗定时器（WDT）使用。因此，AVR 单片机片内的 WDT 是独立硬件形式的看门狗，使用 AVR 单片机可以省掉外部的 WDT 芯片。使用 WDT 可以有效地提高系统的可靠性。

2.3.3 CPU 的工作时序

AVR 单片机 CPU 的工作是由系统时钟直接驱动的，在片内不再进行分频。图 2-7 所示为哈佛结构和快速访问寄存器组的并行指令存取和指令执行时序。CPU 在启动后第 1 个时钟周期 T1 取出第 1 条指令，在 T2 周期便执行取出的指令，并同时又取出第 2 条指令，依次进行。这种基于流水线形式的取指令方式，使 AVR 单片机可以以非常高的速度执行指令，获得高达 1MIPS/MHz 的效率。

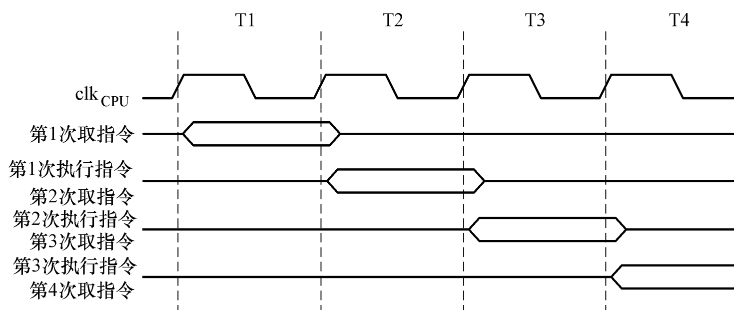


图 2-7 并行指令存取和指令执行

图 2-8 所示为 ALU 与寄存器堆操作单周期指令的执行时序。在单一时钟周期内，由 2 个寄存器提供操作数，ALU 执行相应的操作，最后将操作结果回送到目的寄存器中。

AVR 单片机对片内 SRAM 的访问需要 2 个时钟周期。如图 2-9 所示，在 2 个系统时钟周期内，ALU 完成对片内数据存储器（SRAM）访问的操作时序。

2.3.4 存储器

AVR 单片机在片内集成了 Flash 程序存储器、SRAM 数据存储器和 EEPROM 数据存储器。3 个存储器空间互相独立，物理结构也不同。程序存储器为闪存存储器 Flash，以 16 位（字）为一个存储单元，作为数据读取时，以字节为单位，而擦除、写入则是以页为单位的

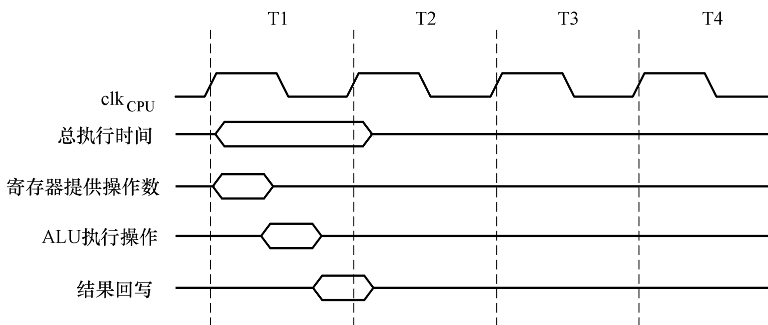


图 2-8　单周期 ALU 操作

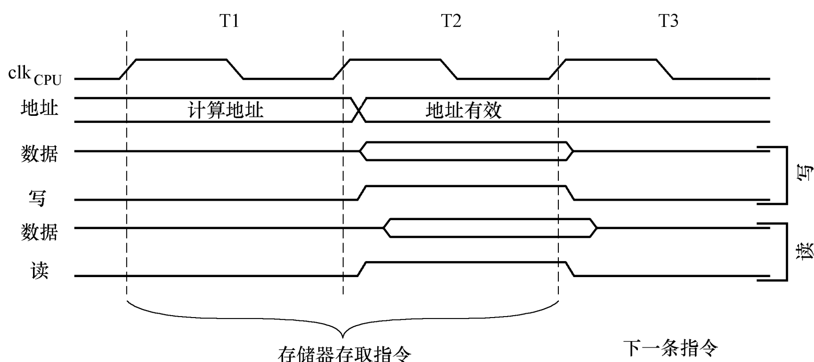


图 2-9　片内 SRAM 访问时序

（不同型号 AVR 单片机一页的大小也不同）。SRAM 数据存储器是以 8 位（字节）为一个存储单元，编址方式采用与工作寄存器组、I/O 寄存器和 SRAM 统一寻址的方式。EEPROM 数据存储器也是以 8 位（字节）为一个存储单元，对其的读写操作都以字节为单位。有关存储器结构将在下一节详细叙述。

2.3.5　I/O 口

ATmega16 有 4 个 8 位的双向 I/O 口 PA、PB、PC、PD，它们对外对应 32 个 I/O 引脚，每一位都可以独立地用于逻辑信号的输入和输出。在 5V 工作电压下，输出时每个引脚可输出达 20mA 的驱动电流。而输入时，每个引脚可吸纳最大为 40mA 的电流，可直接驱动发光二极管（LED）（一般 LED 的驱动电流为 10mA 左右）和小型继电器。

AVR 单片机大部分的 I/O 口都具备双重功能，分别与片内的各种不同功能的外围接口电路组合成一些可以完成特殊功能的 I/O 口，如定时器、计数器、串行接口、模拟比较器、捕捉器等。实际上，学习单片机的主要任务，就是了解、掌握单片机 I/O 口的功能，以及如何正确设计这些端口与外围电路的连接构成一个嵌入式系统，并编程、管理和运用它们完成各种各样的任务。有关这些内容，将在后面的章节中逐步介绍。

2.4　存储器结构和地址空间

2.4.1　支持 ISP 的 Flash 程序存储器

AVR 单片机包括 1K～128KB 的片内支持 ISP 的 Flash 程序存储器。由于 AVR 单片机所

有指令为16位字或32位双字，故 Flash 程序存储器的结构为（512～64KB）×16位。Flash 存储器的使用寿命最少为1万次写/擦循环。ATmega16 单片机的程序存储器为 8KB×16（16KB×8），程序计数器 PC 宽为13位，以此来对 8K 字程序存储器地址进行寻址。

程序存储器的地址空间与数据存储器的地址空间是分开的，地址空间从 $000 开始。如要在程序存储器中使用常量表，则常量表可以被设定在整个 Flash 地址空间中。

2.4.2 SRAM 数据存储器空间

图2-10给出 ATmega16 单片机 SRAM 数据存储器的组织结构。全部共1120个数据存储器地址为线性编址，前96个地址为寄存器组（32个8位通用寄存器）、I/O 寄存器（64个8位 I/O 寄存器），分配在 SRAM 数据地址空间的 $0000～$001F、$0020～$005F。接下来的1024个地址是片内数据 SRAM，地址空间占用 $0060～$045F。

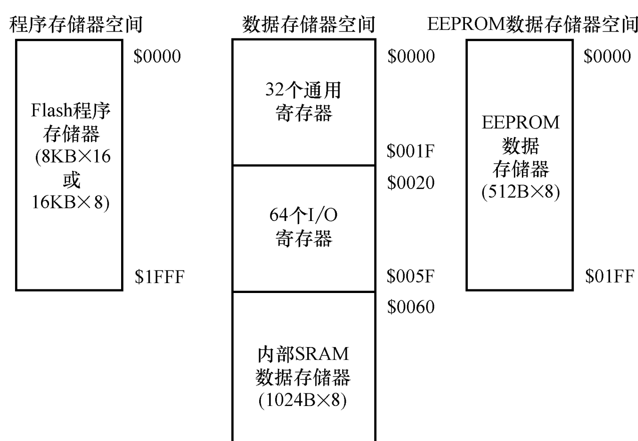


图2-10 ATmega16 存储器结构

CPU 对 SRAM 数据存储器的寻址方式分为5种：直接寻址、带偏移量的间接寻址、间接寻址、带预减量的间接寻址和带后增量的间接寻址。在寄存器堆中，寄存器 R26～R31 具有间接寻址指针寄存器的特性。ALU 可使用直接寻址的方式对整个存储器空间进行寻址操作。带偏移量的间接寻址方式可以寻址由寄存器 Y 和 Z 给出的基本地址附近的63个地址。当使用带预减量和后增量的间接寻址方式时，3个16位的地址寄存器 X、Y 和 Z 都可作为间接寻址的地址指针寄存器，寄存器中的地址指针值将根据操作指令的不同，自动被增加或减小。

32个通用工作寄存器、64个 I/O 寄存器，以及 ATmega16 单片机中 1024B 的数据 SRAM，都可通过上述的寻址方式进行访问操作。

ATmega16 不支持外部 SRAM 扩展，用户使用时要特别注意 SRAM 的容量。

2.4.3 内部 EEPROM 存储器

AVR 系列单片机还包括 64B～4KB 的 EEPROM 数据存储器。它们被组织在一个独立的数据空间中。这个数据空间采用单字节读写方式。EEPROM 的使用寿命至少为10万次写/擦循环。ATmega16 的 EEPROM 容量是 512B，地址范围为 $0000～$01FF。EEPROM 数据存储器可用于存放一些需要掉电保护，而且比较固定的系统参数、表格等。

2.5　通用寄存器组与 I/O 寄存器

全面熟练地理解、掌握 AVR 单片机的通用寄存器组与 I/O 寄存器的性能、特点、功能、设置和使用，是精通和熟练使用 AVR 单片机的关键。由于 AVR 单片机有 32 个通用寄存器和 64 个 I/O 寄存器，其功能、特点及使用方法涉及整个 AVR 单片机的全部功能和特性，因此相对复杂。读者不可能一下就全部掌握它们的应用，只有通过边学习、边实践，逐步深入，加深理解。本节仅给一个总体的介绍，各个不同寄存器具体的使用将在以后相关的章节中加以详细描述。

2.5.1　通用寄存器组

图 2-11 所示为 AVR 单片机中 32 个通用寄存器的结构图。在 AVR 单片机指令集中，所有的通用寄存器操作指令均带有方向，并能在单一时钟周期中访问所有的寄存器。

寄存器名称	对应　SRAM地址	附加功能
R0	$0000	
R1	$0001	
R2	$0002	
⋮		
R13	$000D	
R14	$000E	
R15	$000F	
⋮		
R26	$001A	寄存器 X 低字节
R27	$001B	寄存器 X 高字节

图 2-11　通用寄存器组结构图

用户在使用汇编语言编写程序时，应注意如何正确使用 AVR 单片机中 32 个通用的寄存器。因为这 32 个通用寄存器的功能还是有一定的区别。尤其是 R16 ~ R31 这 16 个寄存器能实现的操作比 R0 ~ R15 要多，如 SBCI、SUBI、CPI、ANDI、ORI 以及 LDI 和乘法指令仅适用于寄存器组中后半部分的寄存器（R16 ~ R31）。另外，R26 ~ R31 还构成 3 个 16 位的地址指针寄存器 X、Y、Z，所以一般情况下不要作为他用。

如图 2-11 所示，每个通用寄存器还被分配在 AVR 单片机的数据存储器空间中，它们直接映射到数据空间的前 32 个地址，因此也可以使用访问 SRAM 的指令对这些寄存器进行访问，但此时在指令中应使用该寄存器在 SRAM 空间的映射地址。通常情况下，最好是使用专用的寄存器访问指令对通用寄存器组进行操作，因为这类寄存器专用操作指令不仅功能强大，而且执行周期也短。

AVR 单片机寄存器组最后的 6 个寄存器 R26 ~ R31 具有特殊的功能，这些寄存器每两个合并成一个 16 位的寄存器，作为对数据存储器空间（使用寄存器 X、Y、Z）以及程序存储器空间（仅使用寄存器 Z）间接寻址的地址指针寄存器。这 3 个间接寄存器 X、Y、Z 由图 2-12 定义。在不同指令的寻址模式下，利用地址寄存器可实现地址指针的偏移、自动增量和减量（参考不同的指令）等不同形式的间

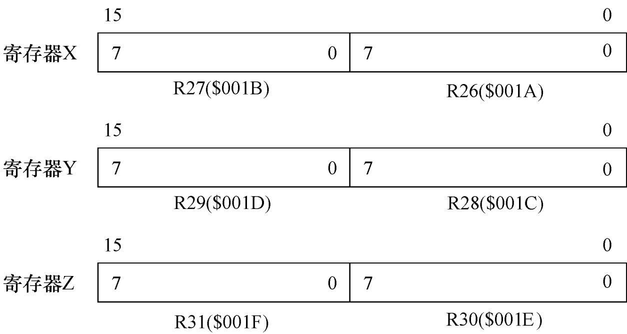


图 2-12　寄存器 X、Y、Z

址寻址操作。

2. 5. 2　I/O 寄存器

表 2-1 列出了 ATmega16 单片机的 I/O 寄存器的地址空间分配、名称和功能。

表 2-1　ATmega16 I/O 寄存器空间分配表

十六进制地址	名　　称	功　　能
$00($0020)	TWBR	TWI 波特率寄存器
$01($0021)	TWSR	TWI 状态寄存器
$02($0022)	TWAR	TWI 从机地址寄存器
$03($0023)	TWDR	TWI 数据寄存器
$04($0024)	ADCL	ADC 数据寄存器低字节
$05($0025)	ADCH	ADC 数据寄存器高字节
$06($0026)	ADCSRA	ADC 控制和状态寄存器
$07($0027)	ADMUX	ADC 多路选择器
$08($0028)	ACSR	模拟比较控制和状态寄存器
$09($0029)	UBRRL	USART 波特率寄存器低 8 位
$0A($002A)	UCSRB	USART 控制状态寄存器 B
$0B($002B)	UCSRA	USART 控制状态寄存器 A
$0C($002C)	UDR	USART I/O 数据寄存器
$0D($002D)	SPCR	SPI 控制寄存器
$0E($002E)	SPSR	SPI 状态寄存器
$0F($002F)	SPDR	SPI I/O 数据寄存器
$10($0030)	PIND	D 口外部输入引脚
$11($0031)	DDRD	D 口数据方向寄存器
$12($0032)	PORTD	D 口数据寄存器
$13($0033)	PINC	C 口外部输入引脚
$14($0034)	DDRC	C 口数据方向寄存器
$15($0035)	PORTC	C 口数据寄存器
$16($0036)	PINB	B 口外部输入引脚
$17($0037)	DDRB	B 口数据方向寄存器
$18($0038)	PORTB	B 口数据寄存器
$19($0039)	PINA	A 口外部输入引脚
$1A($003A)	DDRA	A 口数据方向寄存器
$1B($003B)	PORTA	A 口数据寄存器
$1C($003C)	EECR	EEPROM 控制寄存器
$1D($003D)	EEDR	EEPROM 数据寄存器
$1E($003E)	EEARL	EEPROM 地址寄存器低 8 位
$1F($003F)	EEARH	EEPROM 地址寄存器高 8 位
$20($0040)	UBRRH UCSRC	USART 波特率寄存器高 4 位 USART 状态寄存器 C
$21($0041)	WDTCR	看门狗定时控制寄存器
$22($0042)	ASSR	异步模式状态寄存器
$23($0043)	OCR2	定时器/计数器 2 输出比较寄存器
$24($0044)	TCNT2	定时器/计数器 2(8 位)
$25($0045)	TCCR2	定时器/计数器 2 控制寄存器
$26($0046)	ICR1L	定时器/计数器 1 输入捕捉寄存器低 8 位
$27($0047)	ICR1H	定时器/计数器 1 输入捕捉寄存器高 8 位
$28($0048)	OCR1BL	定时器/计数器 1 输出比较寄存器 B 低 8 位
$29($0049)	OCR1BH	定时器/计数器 1 输出比较寄存器 B 高 8 位
$2A($004A)	OCR1AL	定时器/计数器 1 输出比较寄存器 A 低 8 位
$2B($004B)	OCR1AH	定时器/计数器 1 输出比较寄存器 A 高 8 位
$2C($004C)	TCNT1L	定时器/计数器 1 寄存器低 8 位
$2D($004D)	TCNT1H	定时器/计数器 1 寄存器高 8 位
$2E($004E)	TCCR1B	定时器/计数器 1 控制寄存器 B

（续）

十六进制地址	名　　称	功　　能
$2F（$004F）	TCCR1A	定时器/计数器 1 控制寄存器 A
$30（$0050）	SFIOR	特殊功能 I/O 寄存器
$31（$0051）	OSCCAL OCDR	内部 RC 振荡器校准值寄存器 在线调试寄存器
$32（$0052）	TCNT0	定时器/计数器 0（8 位）
$33（$0053）	TCCR0	定时器/计数器 0 控制寄存器
$34（$0054）	MCUCSR	MCU 控制和状态寄存器
$35（$0055）	MCUCR	MCU 控制寄存器
$36（$0056）	TWCR	TWI 控制寄存器
$37（$0057）	SPMCR	程序存储器写控制寄存器
$38（$0058）	TIFR	定时器/计数器中断标志寄存器
$39（$0059）	TIMSK	定时器/计数器中断屏蔽寄存器
$3A（$005A）	GIFR	通用中断标志寄存器
$3B（$005B）	GICR	通用中断控制寄存器
$3C（$005C）	OCR0	T/C0 计数器输出比较寄存器
$3D（$005D）	SPL	堆栈指针寄存器低 8 位
$3E（$005E）	SPH	堆栈指针寄存器高 8 位
$3F（$005F）	SREG	状态寄存器

AVR 系列单片机所有 I/O 口及外围接口的功能和配置均通过 I/O 寄存器进行设置和使用。CPU 访问 I/O 寄存器可以使用两种不同的方法，使用对 I/O 寄存器访问的 IN、OUT 专用指令，以及使用对 SRAM 访问的指令。

所有的 I/O 寄存器可以通过 IN（I/O 口输入）和 OUT（输出到 I/O 口）指令访问，这些指令是在 32 个通用寄存器与 I/O 寄存器空间之间传输交换数据，指令周期为 1 个时钟周期。此外，I/O 寄存器地址范围在 $00 ~ $1F 之间的寄存器（前 32 个）还可通过指令实现位操作和位判断跳转。SBI（I/O 寄存器中指定位置 1）和 CBI（I/O 寄存器中指定位清零）指令可直接对 I/O 寄存器中的每一位进行位操作。使用 SBIS（I/O 寄存器中指定位为 1 跳行）和 SBIC（I/O 寄存器中指定位为 0 跳行）指令能够对这些 I/O 寄存器中每一位的值进行检验判断，实现跳过一条指令执行下一条指令的跳转。

在 I/O 寄存器专用指令 IN、OUT、SBI、CBI、SBIS 和 SBIC 中使用 I/O 寄存器地址 $00 ~ $3F。

当以 SRAM 方式寻址 I/O 寄存器时，必须将其地址加上 $0020，映射成在数据存储器空间的地址。本书中 I/O 寄存器地址均给出了两种地址表示：I/O 寄存器空间地址以及在数据存储器空间中的映射地址（在圆括号中）。

2.5.3　状态寄存器和堆栈指针寄存器

以下面首先介绍两个在 AVR 单片机中起着非常重要作用的 I/O 寄存器，它们是状态寄存器（SREG）和堆栈指针寄存器（SP）。

1. 状态寄存器（SREG）

状态寄存器（SREG）是一个 8 位标志寄存器，用来存放指令执行后的有关状态和结果的标志。SREG 中各位状态通常是在指令的执行过程中自动形成，但也可以由用户根据需要用专用指令加以改变。

状态标志位的作用很大，每一位都代表着不同的含义。许多指令的运行将对寄存器中的某些位置 1 或清零，它反映了 CPU 运算、操作结果的状态。与状态寄存器（SREG）中的位操作有关的指令有置 1、清零、为 1 转移、为 0 转移等，共有 36 条指令与状态寄存器

（SREG）相关联。由此可见它的重要性。

AVR 单片机的状态寄存器（SREG）在 I/O 空间的地址为 $3F（$005F），其各标志位的意义如下：

位	7	6	5	4	3	2	1	0	
$3F($005F)	I	T	H	S	V	N	Z	C	SREG
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0		0	0	0	0	0	0	

（1）位 7—Ｉ：全局中断使能位

该标志位为 AVR 单片机中断总控制开关，当Ｉ位被置位（“1”）时，表示 CPU 可以响应中断请求，而当Ｉ位被清零（“0”），则所有的中断被禁止，CPU 不响应任何的中断请求。除了该标志位用于 AVR 单片机中断的总控制，各个单独的中断触发控制还由其所在的中断屏蔽寄存器（GIMSK、TIMSK）控制。如果全局中断触发寄存器被清零（“0”），则全局（所有的）中断被禁止，但单独的中断触发控制在 GIMSK 和 TIMSK 中的值保持不变。在中断发生后，I 位由硬件清除，并由 RETI（中断返回）指令置位，从而允许子序列的中断响应。

（2）位 6—T：位复制存储

位复制指令 BLD 和 BST 使用 T 标志位作为源和目标。通用寄存器组任何一个寄存器的一位可以通过 BST 指令被复制到 T 中，而用 BLD 指令则可将 T 中的位值复制到通用寄存器组的任何一个寄存器的一位中。

（3）位 5—H：半进位标志位

半进位标志位 H 表示在一些运算操作过程中有无半进位（低 4 位向高 4 位进、借位）的产生，该标志对于 BCD 码的运算和处理非常有用。

（4）位 4—S：符号标志位，$S = N \oplus V$

S 位是负数标志位 N 和 2 的补码溢出标志位 V 两者的异或值。在正常运算条件下（$V = 0$，不溢出）$S = N$，即运算结果最高位作为符号是正确的。而当产生溢出时 $V = 1$，此时 N 已不能正确指示运算结果的正负，但 $S = N \oplus V$ 还是正确的。对于单（或多）字节有符号数据来说，执行减法或比较操作后，S 标志能正确指示参与相减或比较的两个数的大小。

（5）位 3—V：2 补码溢出标志位

2 的补码溢出标志位 V，支持 2 的补码运算，为模 2 补码加、减运算溢出标志。溢出表示运算结果超过了正数（或负数）所能表示的范围。加法溢出表现为正 + 正 = 负，或负 + 负 = 正；减法溢出表现为正 − 负 = 负，或负 − 正 = 正。溢出时，运算结果最高位（N）取反才是真正的结果符号。

（6）位 2—N：负数标志位

负数标志位直接取自运算结果的最高位，$N = 1$ 时表示运算结果为负，否则为正。但发生溢出时不能表示真实的结果（见上面对溢出标志位的说明）。

（7）位 1—Z：零值标志位

零值标志位表明在 CPU 运算和逻辑操作之后，其结果是否为零，当 $Z = 1$ 时表示结果为零。

（8）位 0 − C：进/借位标志

进/借位标志位表明在 CPU 的运算和逻辑操作过程中有无发生进/借位。

以上这些标志位非常重要，对运算结果的判断处理，要以相应的标志位为依据。标志位也是分支、循环控制的依据。采用汇编语言编写程序时，要注意指令对标志位的影响，以及正确地使用判断指令。

2. 堆栈指针寄存器（SP）

堆栈是数据结构中所使用的专用名词，它是由一块连续的 SRAM 空间和一个堆栈指针寄存器组成，主要应用于快速便捷地保存临时数据、局部变量和中断调用或子程序调用的返回地址。堆栈在系统程序的设计和运行中起着非常重要的作用，只要程序中使用了中断和子程序调用，就必须正确地设置堆栈指针寄存器（SP），在 SRAM 空间建立堆栈区。

堆栈是一种特殊的线性数据结构，数据的进出在堆栈的顶部进行，并遵循后进先出（Last in First out，LIFO）的原则。堆栈指针实际上就是堆栈顶部的地址，它随着堆栈中数据的进出而变化。堆栈指针寄存器（SP）中保存着堆栈指针，即堆栈顶部的地址。

处在 I/O 地址空间 $3E（$005E）和 $3D（$005D）的两个 8 位寄存器构成了 AVR 单片机的 16 位堆栈指针寄存器（SP）。AVR 单片机复位后堆栈寄存器的初始值为 SPH = $00、SPL = $00，因此建议用户程序必须首先对堆栈指针寄存器（SP）进行初始化设置。

位	15	14	13	12	11	10	9	8	
$3E($005E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
$3D($005D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
位	7	6	5	4	3	2	1	0	
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初始化值	0	0	0	0	0	0	0	0	
初始化值	0	0	0	0	0	0	0	0	

AVR 单片机的堆栈区是建立在 SRAM 空间的，16 位的 SP 寄存器可以寻址的空间为 64KB。但在实际应用中，还必须考虑所使用 AVR 单片机芯片 SRAM 空间的实际情况和所配备的 SRAM 容量的大小。首先，堆栈区应该避开寄存器区域所对应的 SRAM 空间，防止堆栈操作时改变了寄存器的设置。由于 AVR 单片机的堆栈是向下增长的，即新数据进入堆栈时栈顶指针的数据将减小（注意：这里与 51 单片机不同，51 单片机的堆栈是向上增长的，即进栈操作时栈顶指针的数据将增加），所以尽管原则上堆栈可以在 SRAM 的任何区域中，但通常初始化时将 SP 的指针设在 SRAM 最高处。

对于具体的 ATmega16 芯片，堆栈指针必须指向高于 $0060 的 SRAM 地址空间，因为低于 $0060 的区域为寄存器空间。ATmega16 片内集成有 1KB 的 SRAM，不支持外部扩展 SRAM，所以堆栈指针寄存器（SP）的初始值应设在 SRAM 的最高端 $045F 处（参考图 2-10）。

AVR 单片机的堆栈有自动硬件进栈（执行调用指令、响应中断）、自动硬件出栈（执行调用返回指令（RET）、执行中断返回指令（RETI））和人工进出栈（进栈（PUSH）、出栈（POP））等指令。AVR 单片机堆栈采用 SP-1 或 SP-2 的进栈操作。

2.6　ATmega16 单片机的工作状态

对采用单片机所构成的嵌入式电子系统来讲，单片机芯片构成了嵌入式系统的“心

脏”，整个系统的正常工作是由单片机来控制、指挥和协调的。可以这样简单地理解：将一块单片机，加上必要的外围电路（如发光二极管、显示器、继电器、按键键盘等），以及根据一定功能要求和硬件电路编写的系统运行程序，三者有机的结合，就能组成各种类型、各种功能、千姿百态的电子系统和产品。如果把电子产品比喻为人，那么单片机就是人的大脑和心脏，外围电路就如同人的五官和四肢，电路板上的路线如同人的神经系统，而系统运行程序就代表人的知识、思维、判断和反应。外围电路通过各种不同的传感器，测量和获取到现实世界的状态（如温度、转速、压力），这些状态值经过线路传送到单片机中，由单片机中的程序进行计算和判断，然后再发出控制信号到外围电路，外围的控制机构产生正确的动作。

AVR 单片机的工作状态通常包括：复位状态、正常程序执行工作状态、休眠节电工作状态、程序运行代码下载的编程，以及熔丝位的配置。用户必须非常熟悉和了解 AVR 单片机的这些工作状态和它们之间的转换关系。

2.6.1 AVR 单片机最小系统

一个单片嵌入式系统的核心，其实就是一个单片机最小系统。它仅仅由一片单片机芯片、两个电阻、一个石英晶体和两个电容构成，如图 2-13 所示。

图 2-13 所示的几个器件所构成的最小系统，就是一颗单片嵌入式系统完整的“心脏”和“大脑”，已经可以工作了。学过 51 单片机的读者可能会发现，这个最小系统里没有复位电路和晶振。是的，在前文中已经讲到，AVR 单片机出厂熔丝配置成内部 RC 1MHz 时钟，当然 AVR 单片机内部还带有上电复位电路，所以最小系统才会这么简单。

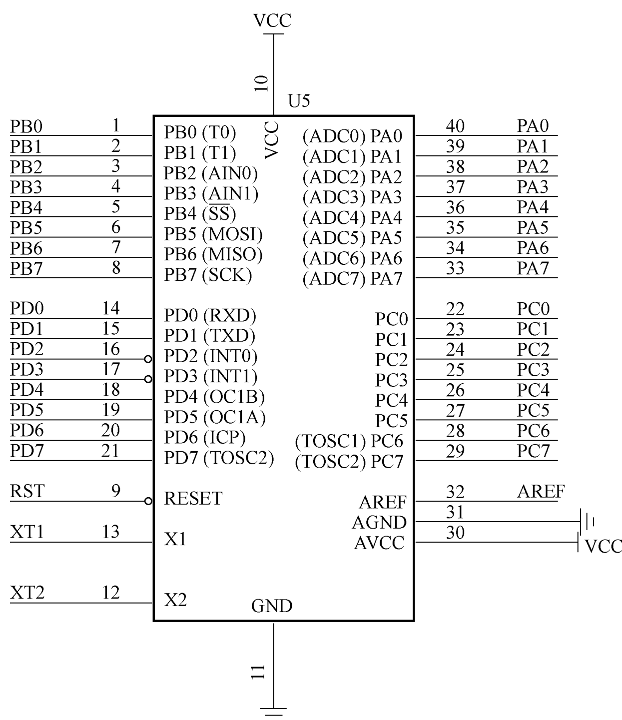


图 2-13　ATmega16 最小系统电路图

2.6.2　AVR 单片机的复位源和复位方式

复位是单片机芯片本身的硬件初始化操作，例如，单片机在上电开机时都需要复位，以便 CPU 以及其他内部功能部件都处于一个确定的初始状态，并从这个初始状态开始工作。

除了系统上电的正常复位初始化之外，当系统程序在运行中出现错误或受到电源的干扰出现错误时，也可通过外部引脚 RESET 进行人工复位，或由芯片内部看门狗定时器（WDT）自动复位，或由芯片内部掉电检测（BOD）来使系统自动进入复位初始化操作。

图 2-14 所示的电路给出了 ATmega16 系统的复位逻辑，表 2-2 给出了复位电特性的参考值。

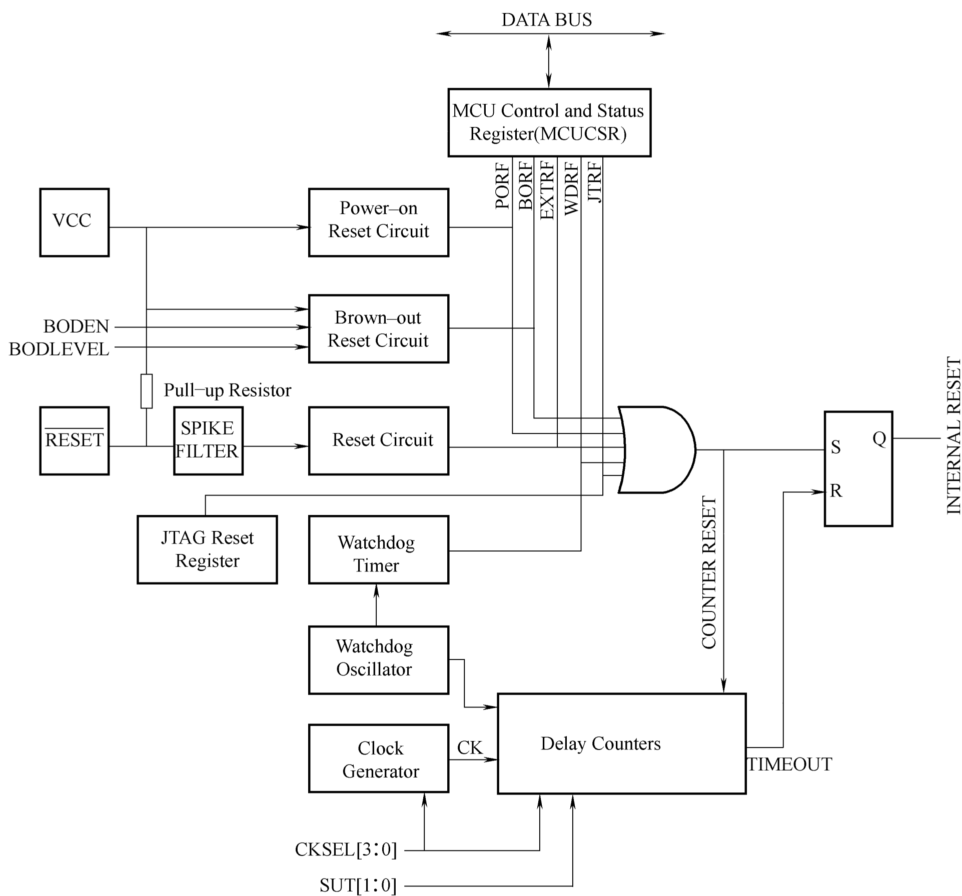


图 2-14　ATmega16 系统复位逻辑图

ATmega16 单片机共有 5 个复位源，如下所示：

1）上电复位。当系统电源电压低于上电复位门限 V_{POT}时，MCU 复位。

2）外部复位。当外部引脚 RESET 为低电平，且低电平持续时间大于 1.5μs 时，MCU 复位。

3）掉电检测（BOD）复位。BOD 使能时，且电源电压低于掉电检测复位门限（4.0V 或 2.7V）时，MCU 复位。

4）看门狗复位。WDT 使能时，并且 WDT 超时溢出时，MCU 复位。

5）JTAG AVR 复位。当使用 JTAG 接口时，可由 JTAG 口控制 MCU 复位。

表 2-2 系统复位电参数

符　号	参　数	条　件	最小值	典型值	最大值	单位
V_{POT}	上电复位门限电压（电源电压上升时）			1.4	2.3	V
	上电复位门限电压（电源电压下降时）			1.3	2.3	V
V_{RST}	RESET 门限电压		0.1*Vcc*		0.9*Vcc*	V
t_{RST}	RESET 最小复位脉冲宽度				1.5	μs
V_{BOT}	BOD 复位门限电压	BODLEVEL = 1	2.5	2.7	3.2	V
		BODLEVEL = 0	3.6	4.0	4.5	
t_{BOD}	BOD 检测的低电压最小宽度	BODLEVEL = 1		2		μs
		BODLEVEL = 0		2		
V_{HYST}	BOD 检测迟滞电压			50		mV

当任何一个复位信号产生时，AVR 单片机将进行复位操作。复位操作过程并不需要时钟源处于运行工作状态（AVR 单片机采用异步复位方式，提高了可靠性）。在 MCU 复位过程中，所有的 I/O 寄存器被设为初始值，程序计数器（PC）置 0。当系统电压高于上电复位门限 V_{POT}（或 BOD 复位门限电压）时，复位信号撤销，硬件系统 Delay Counters 将启动一个可设置的计数延时过程（延时时间为 t_{TOUT}，由一组熔丝位 SUT、CKSEL 确定）。经过一定的延时后，AVR 单片机才进行系统内部真正的复位启动。采用这种形式的复位启动过程，能够保证电源电压在达到稳定后单片机才进入正常的指令操作。

AVR 单片机复位启动后，由于程序计数器（PC）置为 $0000，因此 CPU 取出的第一条指令就是在 Flash 存储器空间的 $0000 处，即复位后系统程序从地址 $0000 处开始执行（指非 BOOT LOAD 方式启动）。通常在 $0000 地址中放置的指令为一条相对转移指令 RJMP 或 JMP，跳到主程序的开始。这样，系统复位启动后，首先执行 $0000 处的跳转指令，然后转到执行主程序的指令。

由此可见，AVR 单片机的复位过程考虑得非常周到，也是非常可靠的。AVR 单片机采用 5 个复位源，异步复位操作，以及内部可设置的延时启动，大大提高了芯片的抗干扰能力和整个系统的可靠性，这在工业控制中非常重要，同时也是 AVR 单片机的优点之一。与此同时，AVR 单片机内部的 MCU 控制和状态寄存器 MCUCSR 还将引起复位的复位源进行了记录，用户程序启动后，可以读取 MCUCSR 中的标记，查看复位是由于何种情况造成的，是正常复位还是异常复位，从而根据实际情况执行不同的程序，实现不同的处理。这对于实现高可靠的系统控制、掉电保护处理、故障处理等应用非常有用，具体请参考 AVR 单片机的器件手册说明。

1. 上电复位

AVR 单片机内部含有上电复位（Power On Reset，POR）电路。POR 电路确保了只有当 V_{CC}超过一个安全电平时，器件才开始工作，如图 2-15、图 2-16 所示。

无论何时，只要 V_{CC}低于检测电平 V_{POT}时，器件进入复位状态。一旦当 V_{CC}超过门限电压 V_{POT}，而 RESET 电压也达到 V_{RST}时，将启动芯片内部的一个可设置的延时计数器。在延时计数器溢出之前，器件一直保持复位状态（保持高电平）。经过 t_{TOUT}时间后，延时计数器溢出，将内部复位信号拉低，CPU 才开始正式工作。

2. 外部复位

外部复位是由外加在 RESET 引脚上的低电平产生的。当 RESET 引脚被拉低于 V_{RST}的时

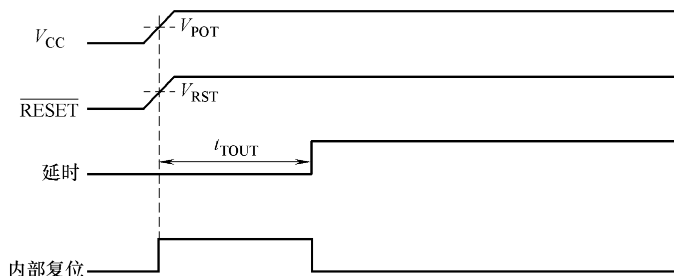


图 2-15　MCU 上电复位启动，RESET 连到 V_{CC}

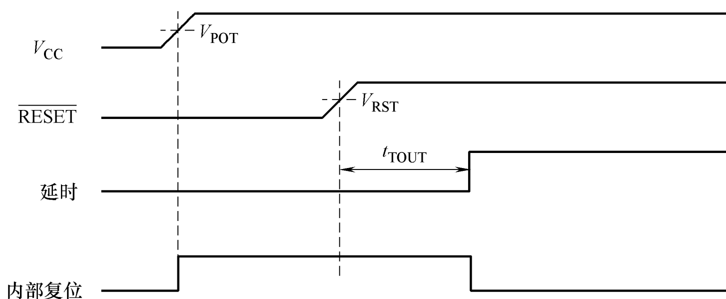


图 2-16　MCU 上电复位启动，RESET 由外部控制

间大于 1.5μs 时即触发复位过程，如图 2-17 所示。当 RESET 引脚电平高于 V_{RST}后，将启动内部可设置的延时计数器。在延时计数器溢出之前，器件一直保持复位状态（保持高电平）。经过 t_{TOUT}时间后，延时计数器溢出，将内部复位信号拉低，CPU 才开始正式工作。

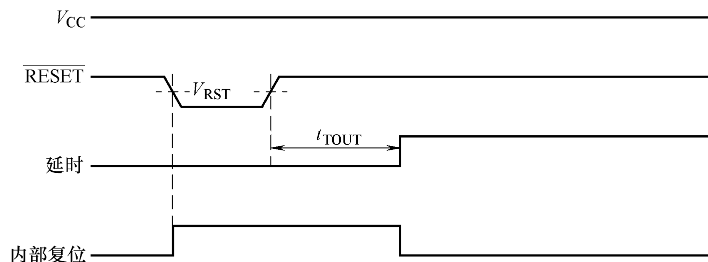


图 2-17　外部 RESET 复位

3. 掉电检测（BOD）复位

ATmega16 有一个片内的 BOD（ Brown Out Detection）电源检测电路，用于在系统运行时对系统电压 V_{CC}的检测，并与一个固定的阈值电压相比较。BOD 检测阈值电压可以通过 BODLEVEL 熔丝位设定为 2.7V 或 4.0V。BOD 检测阈值电压有迟滞效应，以避免系统电源的尖峰毛刺误触发 BOD 检测器。阈值电平的迟滞效应可以理解为上阈值电压$V_{BOT+} = V_{BOT} + V_{HYST}/2$，下阈值电压 $V_{BOT-} = V_{BOT} - V_{HYST}/2$。

BOD 检测电路可以通过编程 BODEN 熔丝位来置成有效或者无效。当 BOD 被置成有效，并且 V_{CC}电压跌到下阈值电压 V_{BOT-}（见图 2-18）以下时即触发复位过程，CPU 进入复位状态。当 V_{CC}回升，而且超过上阈值电压 V_{BOT+}后，再经过设定的启动延时时间，CPU 重新启动运行。注意，只有当 V_{CC}电压低于阈值电压并且持续 t_{BOD}后，BOD 电路才启动延时计数器

计数。

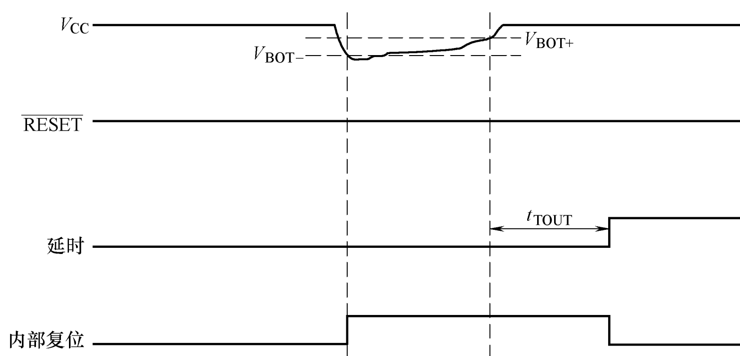


图 2-18　掉电检测（BOD）复位

4. 看门狗复位

ATmega16 片内还集成一个独立的看门狗定时器（WDT）。该 WDT 由片内独立的 1MHz 振荡器提供时钟信号，并且可用专用的熔丝位或由用户通过指令控制 WDT 的启动和关闭，以及设置和清零计数值。当 WDT 启动计数后，一旦发生计数溢出，它将触发产生一个时钟周期宽度的复位脉冲。脉冲的上升沿将使器件进入复位状态，脉冲的下降沿启动延时计数器计数，经过设定的启动延时时间，CPU 重新开始运行（见图 2-19）。使用 WDT 功能，可以防止系统受到干扰而引起的程序运行紊乱和跑飞，提高了系统的可靠性。

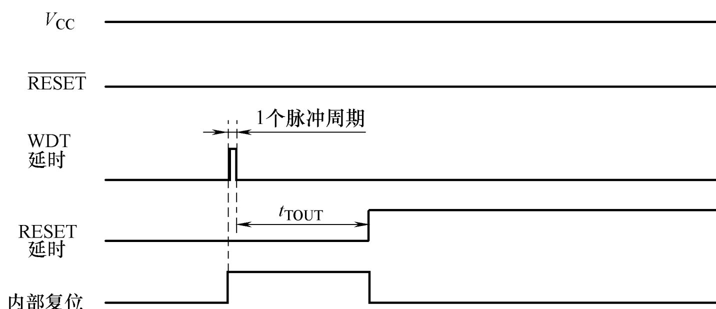


图 2-19　WDT 溢出复位

2.6.3　对 AVR 单片机的编程下载

现在，单片机系统程序的编写、开发和调试都是借助于个人计算机（PC）完成的。用户首先在 PC 上通过使用专用单片机开发软件平台，编写由汇编语言或高级语言构成的系统程序（源程序），再由编译系统将源程序编译成单片机能够识别和执行的运行代码（目标代码）。运行代码的本身是一组二进制的数据，在 PC 中对于纯二进制码的数据文件一般是采用 BIN 格式保存的，以“bin”作为文件的扩展名。但是实际使用中，通常使用的是一种带定位格式的二进制文件（HEX 格式的文件），一般以“hex”作为文件的扩展名。

对单片机的编程操作，通常也称为程序下载，是指以特殊手段和软硬件工具，对单片机进行特殊的操作，以实现下面的 3 种功能：

1）将在 PC 上生成的该单片机系统程序的运行代码写入单片机的程序存储器中。

2）用于对片内的 Flash、EEPROM 进行擦除、数据的写入（包括运行代码）和数据的

读出。

3）实现对 AVR 单片机配置熔丝位的设置、芯片型号的读取、加密位的锁定等。

AVR 单片机支持多种形式的编程下载方式：

（1）高压并行编程方式

对于外围引脚数大于 20 的 AVR 单片机芯片，一般都支持这种高压并行编程方式。这种编程方式也是最传统的单片机的程序下载方式，其优点是编程速度快。但使用这种编程方式需要占用芯片众多的引脚和 12V 的电压，所以必须采用专用的编程器单独对芯片操作。这样 AVR 单片机芯片必须从 PCB 上取下来，不可以实现芯片在线（板）的编程操作，因此这种方式不适合系统调试过程以及产品的批量生产需要。

（2）串行编程方式（ISP）

串行编程方式是通过 AVR 单片机芯片本身的 SPI 或 JTAG 串行口实现的，由于编程时只需要占用比较少的外围引脚，所以可以实现芯片的在系统编程（In System Programmable，ISP），不需要将芯片从 PCB 上取下来，所以串行编程方式也是最方便和最常用的编程方式。

串行编程方式还细分成 SPI、JTAG 方式，前者表示通过芯片的 SPI 串口实现对 AVR 单片机芯片的编程操作，后者则是通过 JTAG 串口来实现的。AVR 单片机的许多芯片都同时集成有 SPI 和 JTAG 两种串口，因此可以同时支持 SPI 和 JTAG 的编程。使用 JTAG 方式编程的优点是，通过 JTAG 串口还可以实现系统的在片实时仿真调试（On Chip Debug），缺点是需要占用 AVR 单片机的 4 个 I/O 引脚。而采用 SPI 方式编程，只需要一根简单的编程电缆，同时可以方便地实现 I/O 口的共用，因此是最常使用的方式。其不足之处是不能实现系统的在片实时仿真调试。

（3）其他编程方式

一些型号的 AVR 单片机还支持串行高压编程方式和 IAP（In Application Programmable，在应用编程）方式。串行高压编程是替代并行高压编程的一种方式，主要针对 8 个引脚的 Tiny 系列的 AVR 单片机使用。IAP 方式则是采用了 ATMEL 公司称为自引导加载（Boot Load）技术实现的，往往在一些需要进行远程修改更新系统程序，或动态改变系统程序的应用中才采用。

ATmega16 片内集成了 16KB 的支持系统在系统编程（ISP）和在应用编程（IAP）的 Flash 程序存储器，以及 512B 的 EEPROM 数据存储器。另外，在它的内部，还有一些专用的可编程单元熔丝位，用于加密锁定和对芯片的配置等。对 ATmega16 编程下载操作，就是在片外对上述的存储器和熔丝单元进行读/写（烧入）以及擦除的操作。

由于 ATmega16 片内含有 SPI 和 JTAG 串口，所以对 ATmega16 能使用 3 种编程的方式：高压并行编程、串行 SPI 编程、串行 JTAG 编程。在本书中将主要介绍和采用串行 SPI 编程方式。

2.6.4　ATmega16 的熔丝位

在 AVR 单片机内部有多组与器件配置和运行环境相关的熔丝位，这些熔丝位非常重要，用户可以通过设定和配置熔丝位，使 AVR 单片机具备不同的特性，以更加适合实际的应用。下面只介绍在开始学习和使用 ATmega16 时，需要特别注意和关心的重要熔丝位的使用配置。

ATmega16 单片机在售出时，片内的 Flash 存储器和 EEPROM 存储器阵列是处在擦除的

状态（即内容为 $FF），且可被编程。同时其器件配置熔丝位的默认值为使用内部 1MHz 的 RC 振荡源作为系统时钟。

1. 存储器加密锁定位

ATmega16 有 2 个加密锁定位 LB1 和 LB2，用于设定对片内存储器的加密方式，用户可在编程方式下，对 LB1、LB2 不编程（1），或编程（0），从而获得对片内存储器不同的加密保护方式，见表 2-3。

表 2-3　加密锁定位保护方式

加密锁定位			保 护 方 式
模 式	LB2	LB1	
1	1	1	无锁定方式(无加密),出厂状态
2	1	0	禁止对 Flash、EEPROM、熔丝位的再编程
3	0	0	禁止对 Flash、EEPROM、加密锁定位、熔丝位的再编程和校验,禁止对加密锁定位、熔丝位的再编程

需要进一步说明的是：

1）在 AVR 单片机的器件手册中，使用已编程（Programmed）和未编程（Unprogrammed）定义加密位和熔丝位的状态。“Unprogrammed”表示熔丝状态为“1”（禁止），“Programmed”表示熔丝状态为“0”（允许），即 1：未编程；0：编程。

2）AVR 单片机的加密位和熔丝位可多次编程，不是 OPT 熔丝。

3）AVR 单片机芯片加密锁定后（LB2/LB1 ＝ 1/0，0/0），在外部不能通过任何方式读取芯片内部 Flash 和 EEPROM 中的数据，但熔丝位的状态仍然可以读取，不能修改配置。

4）需要重新下载程序时，或芯片被加密锁定后，或发现熔丝位配置不对，都必须先在编程状态使用芯片擦除命令，清除芯片内部存储器中的数据，同时解除加密锁定。然后重新下载运行代码和数据，修改和配置相关的熔丝位，最后再次配置芯片的加密锁定位。

5）编程状态的芯片擦除命令是将 Flash 和 EEPROM 中的数据清除，并同时将两位锁定位状态配置成无锁定状态（LB2/LB1 ＝1/1）。但芯片擦除命令并不改变其他熔丝位的状态。

6）下载编程的正确的操作程序是：在芯片无锁定状态下，下载运行代码和数据，配置相关的熔丝位，最后配置芯片的加密锁定位。

2. 系统时钟类型的配置

ATmega16 可以使用多种类型的系统时钟源，最常用的为 2 种：使用内部的 RC 振荡源（1MHz/2MHz/4MHz/8MHz）和外接晶体（晶体可在 0 ~ 16MHz 之间选择）配合内部振荡放大器构成的振荡源。具体系统时钟类型的配置由 CKOPT 和 CKSEL 3. . 0 共 5 个熔丝设定，表 2-4、表 2-5 给出了具体的配置值。用户在使用中，首先要根据实际使用情况进行正确的设置，而且千万注意不要对这些熔丝位误操作。

表 2-4　系统时钟类型为使用内部 RC 振荡源

CKOPT	CKSEL 3. . 0	工作频率范围/MHz
1	0001	1. 0（出厂设定）
1	0010	2. 0
1	0011	4. 0
1	0100	8. 0

AVR 单片机提供了用户更多的灵活选择系统时钟的可能性，以满足和适合实际产品的需要。

ATmega16 在片内集成有内部可校准的 RC 振荡器，能提供固定的 1MHz/2MHz/4MHz/8MHz 的系统时钟，这些频率是在 5V、25℃时的标称数值。CKOPT 和 CKSEL 熔丝按表 2-4 编程配置时，可以选择 4 种内部 RC 振荡源之一作为系统时钟使用，此时将不需要外部的元件。

当产品对系统时钟的精度要求比较高，或需要使用一些特殊频率的系统时钟场合时，如使用了 USART 通信接口，系统时钟频率需要使用 4.6080MHz/7.3728MHz/11.0592MHz 时，就要采用第 2 种方式来组成系统时钟源：使用外接晶体（晶体可在 0 ~ 16MHz 之间选择）配合内部振荡放大器构成振荡源，具体连接电路如图 2-6a 所示。此时需要将 CKOPT 和 CKSEL 熔丝按表 2-5 编程配置。

表 2-5　使用外部晶体与片内振荡放大器构成的振荡源

熔丝位		工作频率范围/MHz	C1、C2 容量/pF（仅适用石英晶振）
CKOPT	CKSEL3..0		
1	101x	0.4 ~ 0.9	仅适合陶瓷振荡器
1	110x	0.9 ~ 3.0	12 ~ 22（应与使用晶体配合）
1	111x	3.0 ~ 8.0	12 ~ 22（应与使用晶体配合）
0	101x，110x，111x	≥1.0	12 ~ 22（应与使用晶体配合）

在表 2-5 中，当 CKOPT = 0 时，振荡器的输出振幅较大，容易起振，适合在干扰大的场合以及使用的晶体超过 8MHz 时的情况下使用。而 CKOPT = 1 时，振荡器的输出振幅较小，这样可以减小对电源的消耗，对外的电磁辐射也较小。

2.6.5　AVR 单片机的工作状态

当 AVR 单片机芯片的 V_{CC} 与系统电源接通后，根据 RESET 引脚的电平值的不同，单片机将进入不同的状态：复位状态、常规工作状态、编程状态。

1. RESET 引脚电平为高

在通常情况下，RESET 引脚通过一个上拉电阻接系统电源，为高电平“1”。在此条件下，一旦接通电源，AVR 单片机将进入上电复位状态。经过短暂的内部复位操作后，芯片便进入了常规的工作状态（BOD 和 WDT 引起的复位类同）。

AVR 单片机处在常规工作状态时，有两种工作方式：正常程序执行工作方式和休眠节电工作方式。

（1）正常程序执行工作方式

正常程序执行工作方式是单片机的基本工作方式。由于硬件的复位操作将程序计数器置为零（PC = $0000），因此程序的执行总是从 Flash 地址的 $0000 开始的（指非 BOOT LOAD 方式启动）。

对于 ATmega16 来讲，Flash 地址的 $0002 ~ $0028 是中断向量区，所以真正实际要开始运行的程序代码一般放在从 $002A 以后的程序地址空间中。标准的做法是在 Flash 的 $0000 单元中放置一条转移指令 JMP 或 RJMP，使得 CPU 在复位重新启动后，首先执行该转移指令，跳过中断向量区，转到执行实际程序的开始处。典型的程序结构如下：

```
Flash 空间地址          指令字              说  明
    $0000             JMP RESET          ;复位中断向量
```

```
         …                    …
         …                    …                          ;向量区
         …                    …
    $002A      RESET:  LDI r16,high(RAMEND)          ;主程序开始
    ……                       ……                          ……
```

（2）休眠节电工作方式

休眠节电工作方式是使单片机处于低功耗节电的一种工作方式。当单片机需要处于长时间等待外部触发信号，待有外部触发后才做相应的处理，或每隔一段时间才需要做处理的情况时，可以使用休眠节电工作方式，以减小对电源的消耗。CPU 处于等待的时候（待机状态）可进入休眠节电工作方式，此时 CPU 暂停工作，不执行任何指令。在休眠节电工作方式中，只有部分单片机的电路处于工作状态，而其他的电路停止工作，这样就可节省单片机对电源消耗，形成系统的省电待机状态。一旦有外部的触发信号，或等待时间到，CPU 从休眠状态中被唤醒，重新进入正常程序执行工作方式。

ATmega16 有 6 种不同的休眠模式，每一种模式对应的电源消耗也不同，被唤醒的方式也有多种类型，用户可以根据实际的需要进行选择。

休眠节电工作方式对使用电池供电的系统非常重要，AVR 单片机提供了更多的休眠模式，更加符合和适应实际的需要。如 ATmega16 处在掉电休眠模式状态，其本身的耗电量小于 1μA。

2. RESET 引脚电平为低

AVR 单片机通电后，如果 RESET 引脚的电平被外部拉为低电平“0”，则芯片将进入和处在复位状态，如图 2-16 和图 2-17 所示。通常情况下，该复位状态一直延续到 RESET 引脚的低电平被撤销。一旦 RESET 恢复了高电平，AVR 单片机将重新启动，进入常规工作状态。利用该特点可以实现对 AVR 单片机系统的人工复位或外部强制复位操作。

尤其需要说明的是，一旦 RESET 引脚的电平被外部拉低，当满足某些特殊条件后，芯片将进入编程状态。例如，如果芯片带有 SPI 接口，支持 SPI 串行编程，则通过以下方式将使芯片进入 SPI 编程状态：

1）外部将 SPI 口的 SCK 引脚拉低，然后外部在 RESET 引脚上施加一个至少为 2 个系统周期以上低电平的脉冲；

2）延时等待 20ms 后，由外部通过 AVR 单片机的 SPI 口向芯片下发允许 SPI 编程的指令。

在 AVR 单片机的器件手册的存储器编程（Memory Programming）一章中串行下载（Serial Downloading）一节里，详细介绍了利用 AVR 单片机的 SPI 接口实现 ISP 编程的硬件连接、编程方式状态的进入过程和串行编程的命令等。

一旦芯片进入编程状态，就可以通过 SPI 口将运行代码写入 AVR 单片机的程序存储器，对片内的 Flash、EEPROM 进行擦除、数据的写入（包括运行代码）和数据的读出，以及实现对 AVR 单片机配置熔丝位的设置、芯片型号的读取和加密位的锁定等操作了。

2.6.6 支持 ISP 编程的最小系统设计

在本节中给出一个最基本的、典型的支持 ISP 编程的 AVR 单片机最小系统硬件图。尽管 ATmega16 的 SPI 和 JTAG 口都可以实现 ISP 在线编程，但采用 SPI 口实现 ISP 在线编程是

最常用的方式，因为这样不会造成 AVR 单片机的 I/O 口浪费。

如图 2-13 以 ATmega16 芯片构成的 AVR 单片机最小系统中，并没有考虑如何实现对 AVR 单片机的编程。如果完全按图 2-13 完成硬件系统后，要对 AVR 单片机编程时，就必须将芯片从 PCB 上取下，放到专用的编程设备上才能将系统的执行代码下载到芯片中，然后再将芯片插回到 PCB 上，对于系统调试和生产都非常不方便。

图 2-20 上增加了一个 ISP 编程下载口，该口的 2、3、4、5 脚与芯片 SPI 口的 MOSI（PB5）、MISO（PB6）、SCK（PB7）和 RESET 引脚连接。当需要改动 AVR 单片机的熔丝位配置，或将编译好的运行代码烧入 AVR 单片机的 FlashROM 中时，就不需要将芯片从 PCB 上取下了。只要将一根简单的编程线插在该编程下载口上，利用 PC 就可以方便地实现上面的操作了。

如 2.6.5 中所介绍的，当 PC 对 AVR 单片机编程时，需要先将 SCK 和 RESET 引脚拉低，使 AVR 单片机芯片进入 SPI 编程状态，然后通过 SPI 口进行下载操作。所以，在设计 AVR 单片机系统硬件时，可考虑使用 SPI 口实现 ISP 的功能。

AVR 单片机的 PB5、PB6、PB7 与编程下载口连接，在编程状态时这 3 个引脚用于下载操作。编程完成拔掉下载线，芯片进入正常工作后，PB5、PB6、PB7 仍可作为普通的 I/O 口或 AVR 单片机的 SPI 口使用，受 AVR 单片机的控制，这是使用 SPI 口实现 ISP 功能的优点之一。需要注意的是，如果系统中使用了这 3 个引脚，PCB 上这 3 个引脚已经与外围器件连接在一起的情况下，就需要对外围的连接情况进行分析。如果外围连接在上电情况时表现为强上拉或强下拉（最极端的情况为接高电平或 GND），那么为了保证 AVR 单片机的 SPI 功能正常工作，应该如图 2-20 所示，串入 3 个隔离电阻，阻值在 2kΩ 左右。

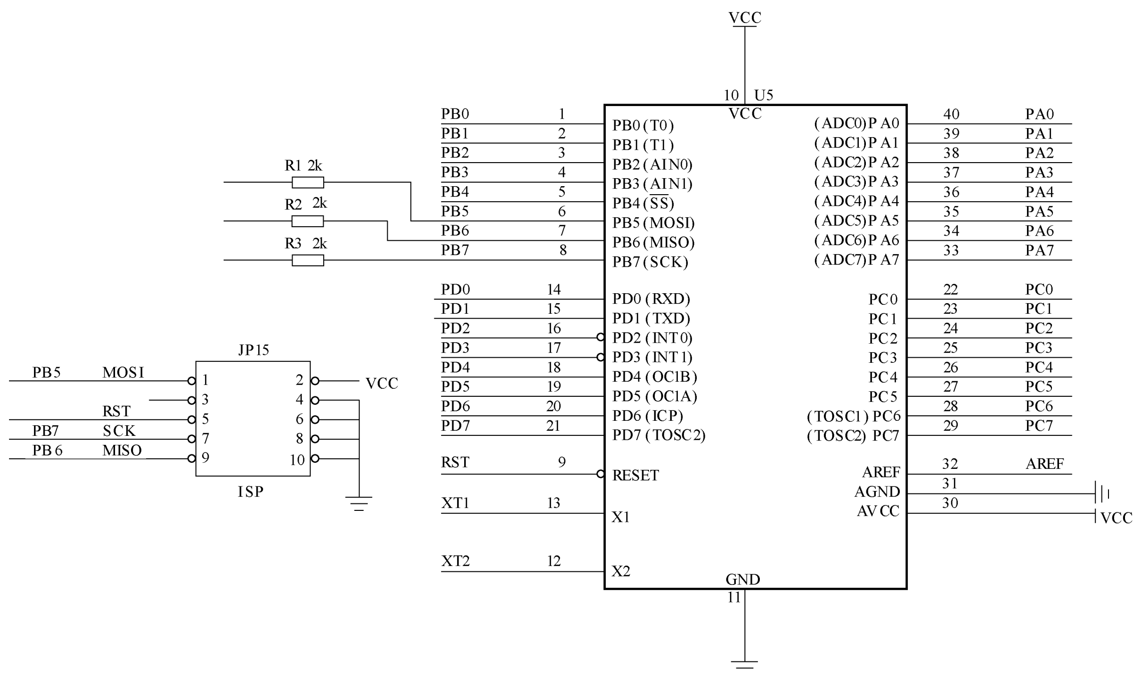


图 2-20　支持 ISP 编程的最小系统设计

对于不同的 AVR 单片机芯片，使用 SPI 方式进行下载编程的硬件连接口，操作命令和时序请参考与该器件相关器件手册中的详细说明，会有所变化（如 ATmega128），但基本方式相同。与使用其他类型的单片机（如 8051）一样，可以采用专用的写入设备进行编程下载，但 AVR 单片机提供了更方便的在线（ISP）串行下载的方法，用户只要制作一个简单的带隔离电路的下载线，就可直接使用 PC 的打印机口实现 AVR 单片机的 Flash、EEPROM 以及熔丝配置位进行编程操作。

2.7 AVR 单片机内部资源的扩展和删减

以上以 ATmega16 为典型介绍了 AVR 单片机的基本结构、性能和特点。ATMEL 公司为满足不同的需求，在 AVR 单片机基本内核的基础上，对芯片的内部资源进行了相应的扩展和删减，形成了 AVR 单片机几十种型号的芯片。尽管它们的功能和外部的引脚定义和封装各有不同，小到 8 ~ 12 个引脚，多到 100 个引脚，但它们的内核结构相同，指令相容，用户可根据实际需要进行选择。

第 3 章　AVR 单片机开发工具安装及开发环境的使用

3.1　AVR Studio 集成开发环境简介及其安装

AVR Studio 集成开发环境是 ATMEL 公司推出的，专门用于开发该公司 AVR 单片机的开发软件平台。它是一个完全免费的，基于 AVR 汇编语言的集成开发环境。AVR Studio 包括了 AVR Assembler 编译器、AVR Studio 软件模拟调试功能、AVR ISP 串行下载和 JTAG 下载功能及 JTAG ICE 在线仿真调试功能。要使用该软件的下载功能和在线仿真调试功能，还需要购买该软件支持的专用仿真下载硬件设备。

AVR Studio 的安装步骤如下：

打开配套光盘内的 AVR Studio 安装文件，双击“setup. exe”进入安装界面（见图3-1）。单击“Next”按钮后出现安装协议说明，选择同意安装协议（见图 3-2）。再单击“Next”按钮出现安装路径的提示，可使用默认方式将其安装在 C 盘。单击“Next”按钮进行文件复制（图 3-3），直到安装完成（见图 3-4）。可在桌面上生成一个快捷方式，以后只需双击快捷图标即可使用。

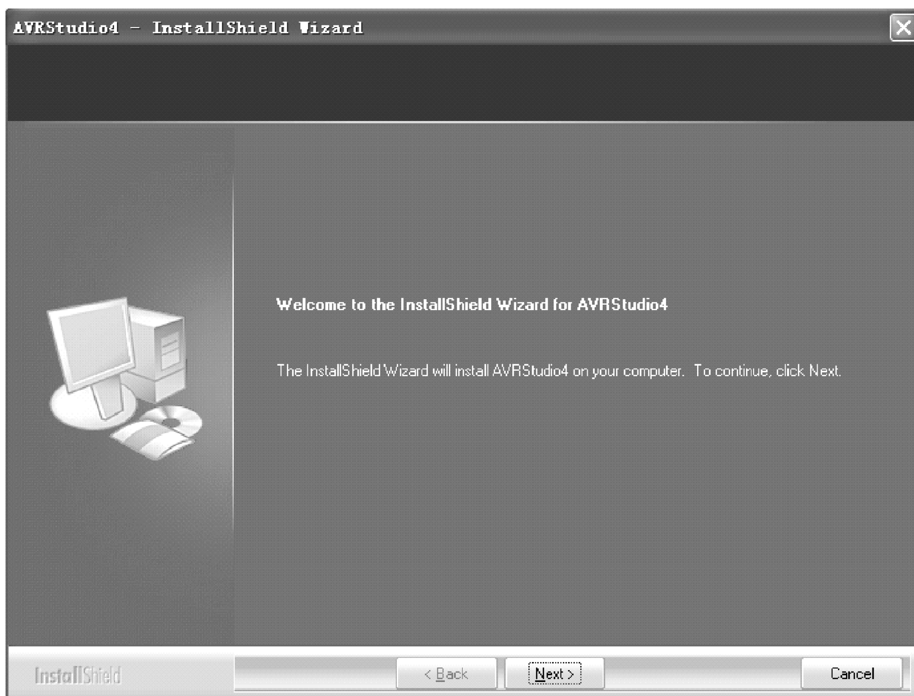


图 3-1　进入安装界面

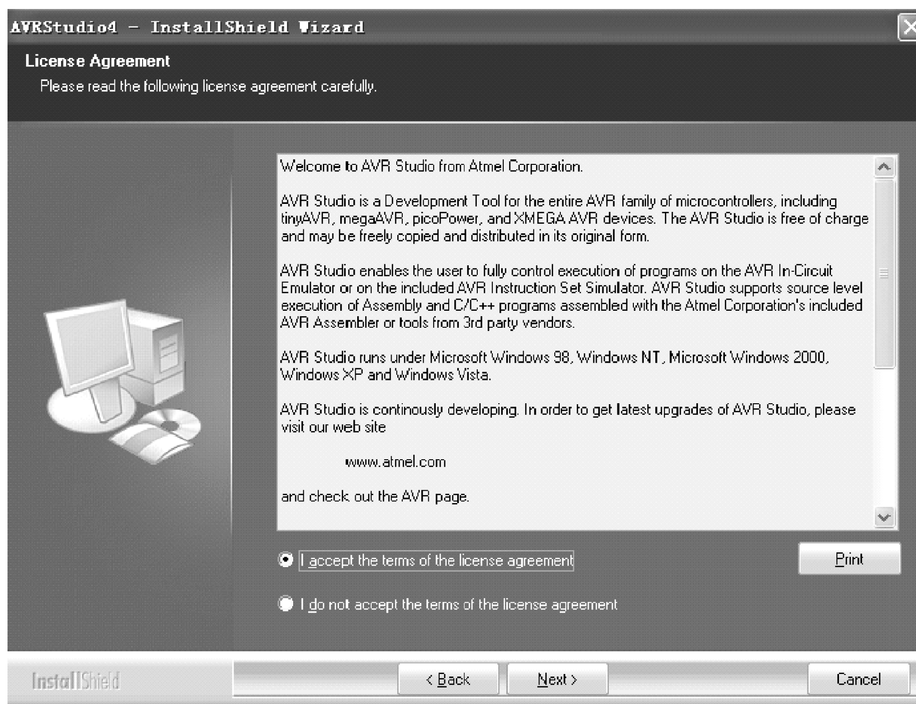


图 3-2　选择同意安装协议

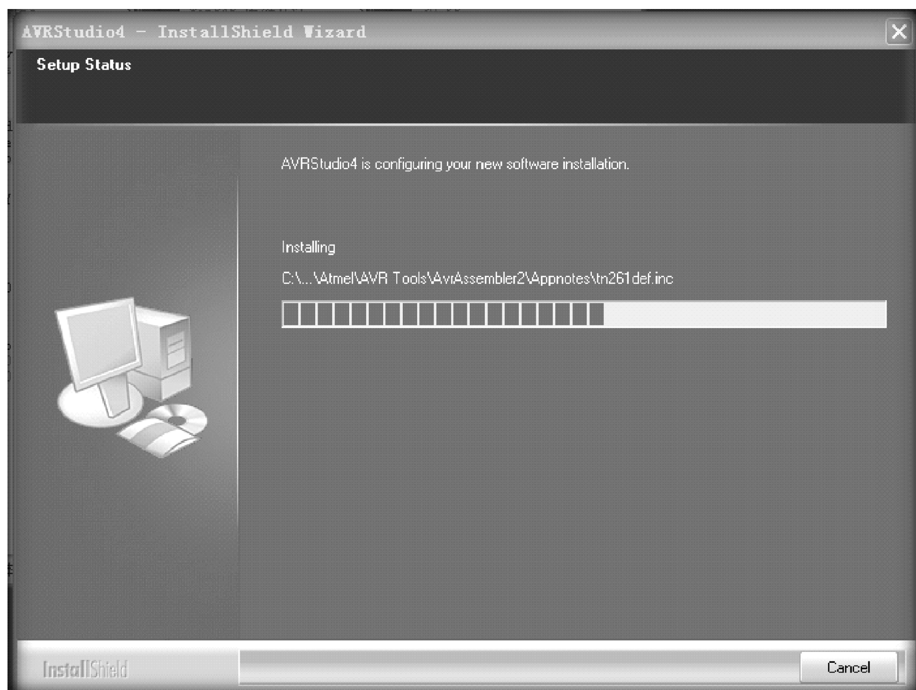


图 3-3　文件复制

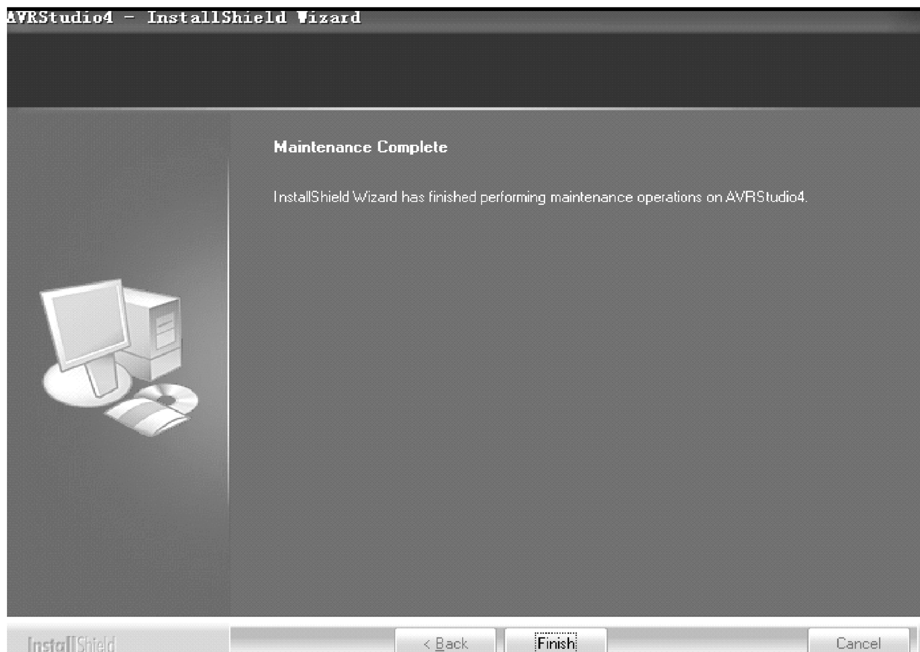


图 3-4　安装完成

3.2　AVR Studio 集成开发环境的使用

3.2.1　建立一个新的工程项目管理文件

启动 AVR Studio 4，AVR Studio 启动后，将看到一个对话框，如图 3-5 所示。需要创建一个新的项目，点击“Create New Project”按钮。



图 3-5　启动后的欢迎对话框

配置项目参数。这个步骤包括选择要创建什么类型的项目、设定名称以及存放的路径。这个过程包括 5 个步骤，如图 3-6 所示：

1）在对话框左边选中“Atmel AVR Assembler”，表明要创建一个项目。

2）输入项目的名称。项目的名称可以随意定义，在例子中用了“Leds”。

3）需要 AVR Studio 自动产生一个汇编文件，在例子中用了“Leds”。

4）选择要存放项目的路径。

5）确认所有的选项，确认之后，按“Next”按钮。

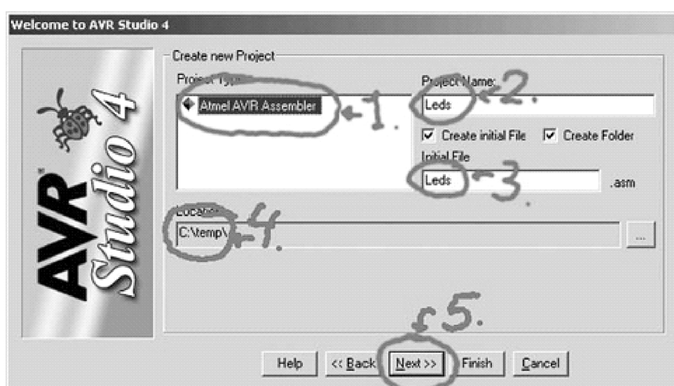


图 3-6　新工程建立对话框

选择调试平台和使用的芯片型号：

1）AVR Studio 4 允许可以选择多种开发调试工具，在这里选用带有仿真功能的 AVR Simulator。

2）芯片选用 ATmega16。

3）检查所有的选项，确认之后，单击“Finish”按钮。

3.2.2　汇编源文件的建立

经过上面的步骤，AVR Studio 打开了一个空的文件，文件的名字是 Leds. asm（见图 3-7）。在新的源文件编辑窗口中输入汇编源程序（见图 3-8）。

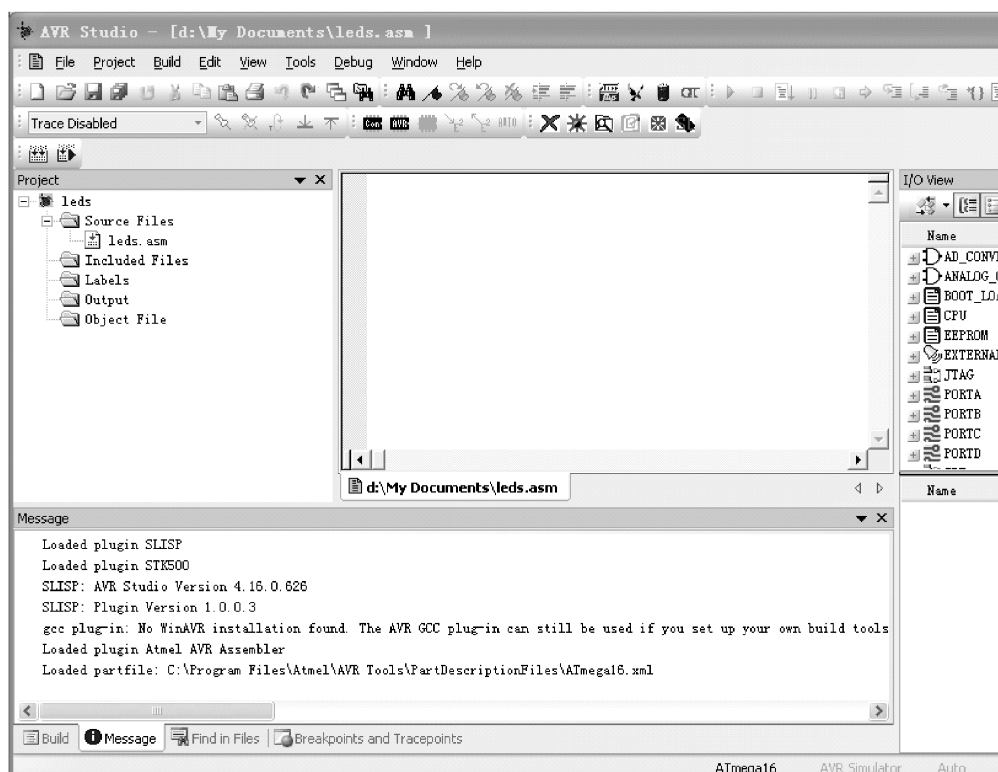


图 3-7　AVR Studio 主工作界面

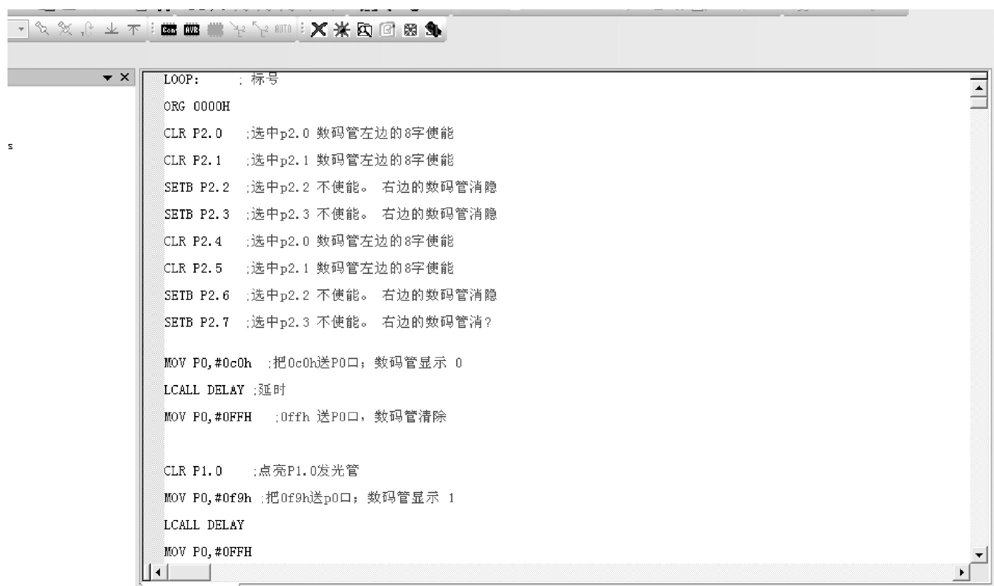


图 3-8 输入汇编源程序

3.2.3 汇编源文件的编译

1）选择菜单项 Build，或使用快捷键 F7，或单击工具栏上对应的工具按钮，对汇编源文件进行编译（见图 3-9）。

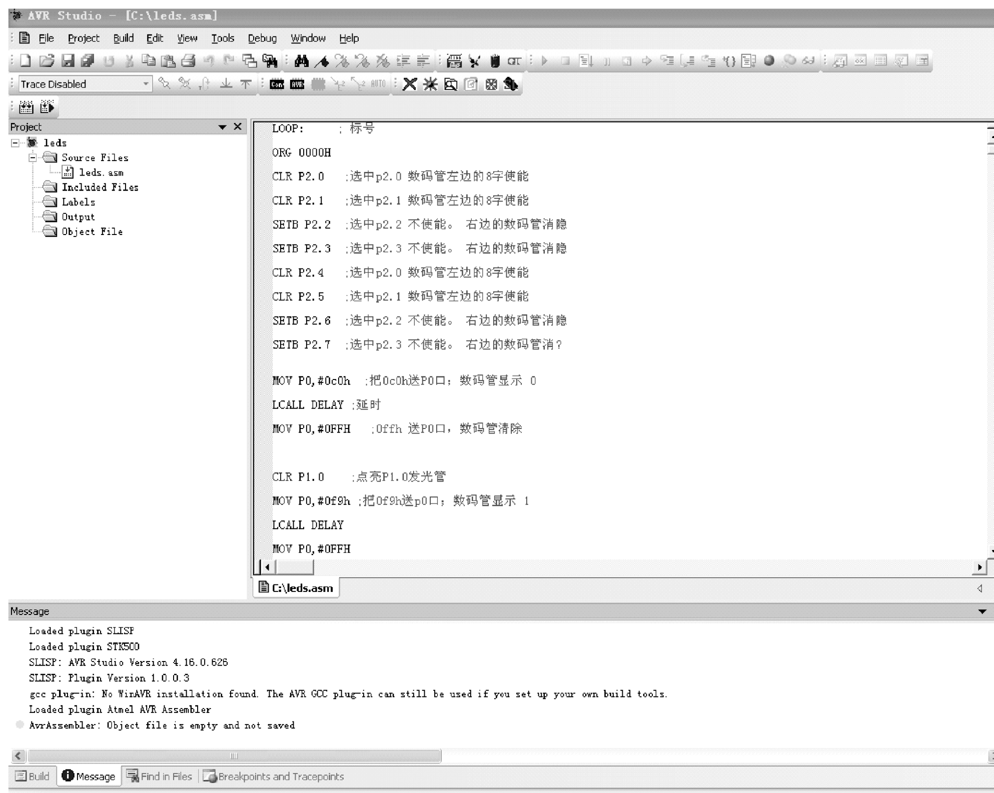


图 3-9 编译源文件

2）编译结束后，AVR studio 将在 Build 的信息提示窗中显示编译结果。如果发现提示中给出了编译过程产生语句错误，则用户应该对源程序进行改正后重新编译，一直到编译结果正确为止。

3.3　ICCAVR 集成开发环境简介

自 ATMEL 公司的 AT90 系列单片机诞生以来有很多第三方厂商为 AT90 系列开发了用于程序开发的 C 语言工具，ICCAVR 就是 ATMEL 公司推荐的第三方 C 编译器之一。ICCAVR 是一种符合 ANSI 标准的 C 语言来开发 MCU 程序的工具，功能合适、使用方便、技术支持好，它主要有以下几个特点：

1）ICCAVR 是一个综合了编辑器和工程管理器的集成工作环境（IDE）；

2）源文件全部被组织到工程之中，文件的编辑和工程的构建也在这个环境中完成，错误显示在状态窗口中，并且当点击编译错误时，光标自动跳转到错误的那一行；

3）工程管理器还能生成可以直接使用的 INTEL HEX 格式文件，该格式的文件可被大多数编程器所支持，用于下载到芯片中；

4）ICCAVR 是一个 32 位的程序，支持长文件名。

3.3.1　ICCAVR 编译器的安装

打开配套光盘内的 ICCAVR 安装文件，双击“setup. exe”进入安装界面（见图 3-10）。单击“下一步”按钮后出现安装协议说明，选择同意安装协议。再单击“下一步”按钮出现安装路径的提示（见图 3-11），可使用默认方式将其安装在 C 盘。单击“下一步”按钮进行文件复制（见图 3-12），直到安装完成（见图 3-13）。可在桌面上生成一个快捷方式，以后只需双击快捷图标即可使用。



图 3-10　进入安装界面

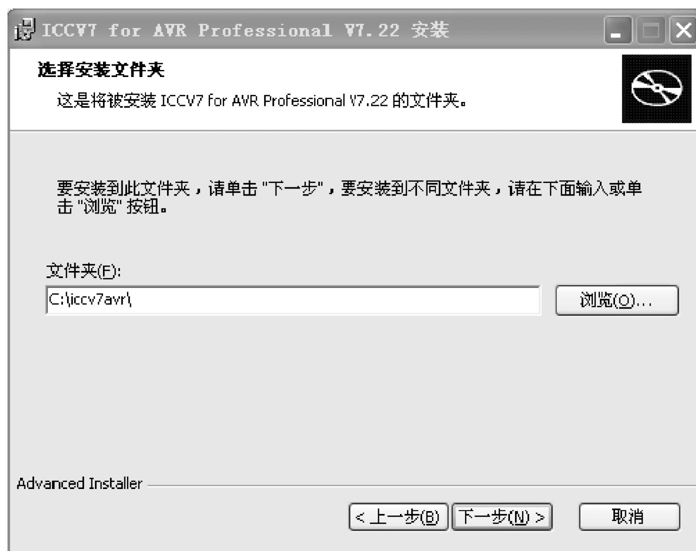


图 3-11　安装路径提示（默认 C 盘）



图 3-12　文件复制

注意：

1）按上述方法进行安装后，得到的是一个只可以使用 30 天的未注册版，对正式版用户还要进行第 2 步的注册才可得到一个无时间限制的正式版。

2）ICCAVR 正式版分标准版和专业版，在标准版中有一些功能限制，如代码的压缩，工程和文件的配置检查，在标准版中不可以使用。

对安装完成的软件进行注册的步骤如下：

对首次安装并且使用期未超过 30 天的用户可以这样注册：

1）启动 ICCAVR 编译器的集成工作环境（IDE）。

2）将正式版中附带的一张名称为“Unlock Disk”的软盘插入机器的软盘驱动器中。

3）在 IDE 的 Help 菜单中寻找标题为“Importing a License from a Floppy Disk”的一项并

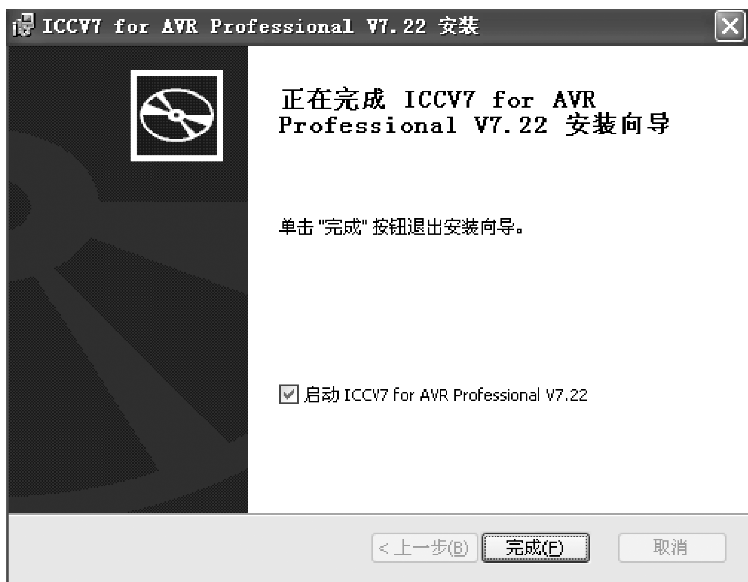


图 3-13　安装完成

且单击。

4）ICCAVR 软件自动进行注册，当注册完成后会提示注册文件已从软盘移走，当确定并再次重新启动 ICCAVR 后会发现软件已经完成注册。

对不是首次安装或使用时间已超过 30 天的用户可这样注册：

1）这类用户在程序启动时已不能进入 IDE，而是出现一个提示注册的对话框，应该选择“Yes”按钮。

2）这时会出现一个注册对话框，对话框上有一个标题为“Importing a License from a Floppy Dis”的按钮。

3）将正式版中附带的一张名称为“Unlock Disk”的软盘插入机器的软盘驱动器中，单击上一步中提到的按钮。

4）ICCAVR 软件自动进行注册，当注册完成后会提示注册文件已从软盘移走，当确定并再次重新启动 ICCAVR 后会发现软件已经完成注册。

注意：“Unlock Disk”软盘在注册时应打开写保护，否则无法完成注册。完成注册后“Unlock Disk”软盘成为一张空盘，不可以在另一台机器上进行安装和注册。当需要在不同的电脑中使用 ICCAVR 或在同一台电脑中将 ICCAVR 重新安装在与原来不同的目录位置时，应该首先在 Help 菜单中选择“Transferring Your License to a Floppy Disk”一项将注册文件传送到一张软盘上，然后再按上述方法进行安装注册。

3.3.2　ICCAVR 中的文件类型及其扩展名

文件类型是由它们的扩展名决定的，IDE 和编译器可以使用以下几种类型的文件。

1. 输入文件

. c——表示是 C 语言源文件；

. s——表示是汇编语言源文件；

. h——表示是 C 语言的头文件；

. prj——表示是工程文件，这个文件保存由 IDE 所创建和修改的一个工程的有关信息；

. a——表示是库文件，它可以由几个库封装在一起，libcavr. a 是一个包含了标准 C 的库和 AVR 特殊程序调用的基本库。如果库被引用，链接器会将其链接到用户的模块或文件中，用户也可以创建或修改一个符合需要的库；

2. 输出文件

. s——表示是每个 C 语言源文件，由编译器在编译时产生的汇编输出文件；

. o——表示汇编文件汇编产生的目标文件，多个目标文件可以链接成一个可执行文件；

. hex——表示是 INTEL HEX 格式文件，其中包含了程序的机器代码；

. eep——表示是 INTEL HEX 格式文件，包含了 EEPROM 的初始化数据；

. cof——表示是 OFF 格式输出文件，用于在 ATMEL 的 AVR Studio 环境下进行程序调试；

. lst——表示是列表文件，在这个文件中列举出了目标代码对应的最终地址；

. mp——表示是内存映像文件，它包含了程序中有关符号及其所占内存大小的信息；

. cmd NoICE 2. xx——表示是调试命令文件；

. noiNoICE 3. xx——表示是调试命令文件；

. dbgImageCraft——表示是调试命令文件；

3.3.3　ICCAVR 的附注和扩充

#pragma（编译附注）；这个编译器接受以下附注：

#pragma interrupt_handler <func1>：<vector number> <func2>：<vector> ...

这个附注必须在函数之前定义。它说明函数 func1、func2 是中断操作函数。所以编译器在中断操作函数中生成中断返回指令 reti 来代替普通返回指令 ret，并且保存和恢复函数所使用的全部寄存器，同样编译器根据中断向量号 vector number 生成中断向量地址。

#pragma ctask <func1> <func2>...

这个附注指定了函数不生成挥发寄存器来保存和恢复代码，它的典型应用是在 RTOS 实时操作系统中让 RTOS 核直接管理寄存器。

#pragma text：<name>

这个附注改变代码段名称，使其与命令行选项相适应。

#pragma data：<data>

这个附注改变数据段名称，使其与命令行选项相适应。这个附注在分配全局变量至 EEPROM 中时必须被使用，读者可参考访问 EEPROM 的例子。

#pragma abs_address：<address>

这个附注表示函数与全局数据不使用浮动定位（重定位），而是从 <address> 开始分配绝对地址。这在访问中断向量和其他硬件项目时特别有用。

#pragma end_abs_address

这个附注结束绝对定位，使目标程序使用正常浮动定位。

C++　注释

如果选择了编译扩充（Project->Options->Compiler），可以在源代码中使用 C++的//类型的注释。

二进制常数

如果选择了编译扩充（Project- > Options- > Compiler），可以使用 0b < 1 | 0 > * 来指定二进制常数。例如，0b10101 等于十进制数 21。

在线汇编

可以使用 asm（"string"）函数来指定在线汇编代码。读者可参考在线汇编。

3.3.4 ICCAVR 的代码转换

IAR C 编译器作为应用于 AVR 的第一个 C 编译器，它有十分丰富的源代码。当从 IAR 编译系统转换到 ImageCraft 编译系统时，绝大多数符合 ANSI C 标准的程序代码不需要转换，IAR C 中 IO 寄存器的定义与 ICCAVR 也是相同的。下面有几个对照的例子。

对于中断操作描述，ICCAVR 使用 pragma 附注描述中断操作函数，而 IAR 引入了语法扩充（interrupt 关键字）。在 ICCAVR 中：#pragma interrupt_ handler func：4//4 是这个中断的向量号，func 为中断处理函数名称，ICCAVR 可以使多个中断向量共用一个中断处理函数；在 IAR 中：interrupt ［vector_ name］func（ ）// vector_ name 是某一个中断向量的名称，IAR C 的中断向量地址使用中断名称来代替，以增加程序的可读性。

对于扩充关键字，IAR 引入 flash 关键字将项目分配进入程序存储空间 FLASH 存储器，ICCAVR 使用 const 关键字来达到相同的目的。

对于过程调用转换，在两个编译系统之间函数参数传递使用的寄存器是不同的，仅影响手工写的汇编函数。

对于在线汇编、宏等，IAR 不支持在线汇编符号，而 ICCAVR 支持在线汇编。

3.4 ICCAVR 向导

1. 起步

自启动 IDE 后，首先从“Project”菜单系统选择“Open”命令，进入“\icc\examples. avr”目录并且选择并打开“led”工程，工程管理器显示在这个工程中只有一个文件 led. c。然后从“Project”菜单中选择“Options”命令打开工程编译选项，在“Target”标号下选择目标处理器，然后从“Project”菜单中选择“Make Project”命令，IDE 将调用编译器编译这个工程文件，并且在状态窗口中显示所有的信息。

如果没有错误，在与源文件同一个目录（在这个例子中是 \icc\examples. avr）中输出一个文件 led. hex。这个文件是 INTEL HEX 格式，大多数能支持 AVR MCU 的编程器和模拟器都支持这种格式，并且能下载这个程序进入目标系统，这样就完成了一个程序的构筑。

如果希望用支持 COFF 调试信息的工具来测试程序，比如 AVR Studio，那么需要从“Project”菜单中选择“Options”命令在编译标签下选择“COFF”输出文件格式。对一些常用的功能，也可使用工具条或鼠标右键弹出菜单。例如，可以在工程窗口单击鼠标右键选择编译选项。

在工程窗口中双击文件名，IDE 将使用编辑器打开这个文件。按这个方法打开 led. c，作为试验可设置一些错误，例如从一行中删除分号“;”。现在从“Project”菜单中选择“Make Project”命令，IDE 首先自动保存已经改变的文件，且开始编译这个文件。这时在状态窗口中会显示错误信息，单击状态窗口中错误信息行或单击其左边的错误符号，光标将移到编辑器中错误行的下面一行上（基本上所有 C 编译器都是这样）。

开始一个新的工程：

从“Project”菜单中选择“New”命令，并且浏览至希望输出工程文件的目录，输出文件的名称取决于工程文件名称。例如，如果创建一个名称为“foo. prj”的工程，那么输出文件名称为“foo. hex”或“foo. cof”等。

自从创建工程后，可以开始写源代码（C 或汇编格式），且将这个文件加入到工程文件排列中。单击工具栏中“Build”图标，可以很容易地构筑这个工程，IDE 输出与 ATMEL 的 AVR Studio 完全兼容的 COFF 文件，可以使用 ATMEL 的 AVR Studio 来调试代码。

为更容易地使用这个开发工具，可以使用应用程序向导来生成一些使用有关硬件的初始化代码。

2. C 程序的剖析

一个 C 程序必须定义一个 main 调用函数，编译器会将你的程序与启动代码和库函数链接成一个“可执行”文件，因此也可以在目标系统中执行它。启动代码的用途在启动文件中很详细地被描述了，一个 C 程序需要设定目标环境，启动代码初始化这个目标使其满足所有的要求。

通常 main 例程完成一些初始化后，然后是无限循环地运行。作为例子，让我们看 \icc\examples目录中的文件 led. c。

```
#include <io8515. h>
/*  为使能够看清 LED 的变化图案，延时程序需要有足够的延时时间 */
void Delay( )
{
unsigned char a,b;
for( a =1; a; a + + )
  for( b =1; b; b + + )
  ;
}
void LED_On( int i)
{
PORTB = ~BIT(i);  /*  低电平输出使 LED 点亮 */
Delay( );
}
void main( )4
{
int i;
DDRB =0xFF;/*定义 B 口输出 */
PORTB =0xFF;/* B 口全部为高电平,对应 LED 熄灭 */
while(1)
  {
  /* LED 向前步进 */
  for(i =0;i <8;i + + )
  LED_On(i);
```

```
/ * LED 向后步进 * /
for(i = 8;i > 0;i - - )
LED_On(i);
/ * LED 跳跃 * /
for(i = 0;i < 8;i + = 2)
LED_On(i);
for(i = 7;i > 0;i - = 2)
LED_On(i);
}
}
```

整个 main 例程是很简单的，在初始化一些 IO 寄存器之后，它运行在一个无限循环中，并且在这个循环中改变 LED 的步进图案。LED 是在 LED_ On 例程中被改变的，LED_ On 例程中直接写正确的数值 IO 端口。因为 CPU 运行很快，为能够看见图案变化，LED_ On 例程调用了延时例程。因为延时的实际延时值不能被确定，这一对嵌套循环只能给出延时的近似延时时间；如果这个实际定时时间是重要的，那么这个例程应该使用硬件定时器来完成延时。

其他的例子，8515intr. c 程序很简单，但同样清楚地显示了如何用 C 写一个中断处理过程，这两个例子可以作为程序的起点。

3. 5　ICCAVR 的 IDE 环境

1. 编译一个单独的文件

正常建立一个输出文件的次序是，首先应该建立一个工程文件并且定义属于这个工程的所有文件。然而，有时也需要将一个文件单独地编译为目标文件或最终的输出文件。这时可以这样操作，从 IDE 菜单 File 中选择“Compile File. . . ”命令，来执行“to Object”和“to Output”中的任意一个。当调用这个命令时，文件应该是打开的，并且在编辑窗口中是可以编辑的。

编译一个文件为目标文件（to Object)，对检查语法错误和编译一个新的启动文件是很有用的。编译一个文件为输出文件（to Output)，对较小的并且是一个文件的程序较为有用。注意：这里使用默认的编译选项。

2. 创建一个新的工程

为创建一个新的工程，从菜单“Project”中选择“New”命令，IDE 会弹出一个对话框，在对话框中可以指定工程的名称，这也是输出文件的名称。如果使用一些已经建立的源文件，可在菜单“Project”中选择“AddFile（s)”命令。

另外，可以在菜单“File”中选择“New”命令来建立一个新的源文件来输入代码。可以在菜单“File”中选择“Save”或“Save As”命令来保存文件，然后可以像上面所述调用“AddFile（s)”命令将文件加入到工程中，也可在当前编辑窗口中单击鼠标右键选择“Add to Project”将文件加入已打开的工程列表中。通常输出源文件在工程同一个目录中，但也可不作这样要求。

工程的编译选项使用菜单“Project”中的“Options”命令。

3. 工程管理

工程管理允许将多个文件组织进同一个工程，而且定义它们的编译选项。这个特性允许将工程分解成许多小的模块，当处理工程构筑时，只有一个文件被修改和重新编译，如果一个文件作了修改，编译包含这个头文件的源文件时，IDE 会自动重新编译已经改变的头文件。

一个源文件可以写成 C 或汇编格式的任意一种。C 文件必须使用“. c”扩展名，汇编文件必须使用“. s”扩展名。可以将任意文件放在工程列表中，例如可以将一个工程文档文件放在工程管理窗口中，工程管理器在构筑工程时对源文件以外的文件不予理睬。

对目标器件不同的工程，可以在编译选项中设置有关参数。当新建一个工程时，使用默认的编译选项，可以将现有编译选项设置成默认选项，也可将默认编译选项装入现有工程中。默认编译选项保存在 default. prj 文件中。

为避免工程目录混乱，可以指定输出文件和中间文件到一个指定的目录，通常这个目录是工程目录的一个子目录。

4. 编辑窗口

编辑窗口是与 IDE 交流信息的主要区域，在这个窗口中可以修改相应的文件，当编译存在错误时，用鼠标单击有关错误信息时，编辑器会自动将光标定位在错误行的位置。注意：对 C 源文件中缺少分号的错误编辑器定位于其下面一行。

5. 应用构筑向导

应用构筑向导是用于创建外围设备初始化代码的一个图形界面，可以单击工具条中的“Wizard”按钮或菜单“Tools”中的“ApplicationBuilder”命令来调用它。

应用构筑向导使用编译选项中指定的目标 MCU 来产生相应的选项和代码。

应用构筑向导显示目标 MCU 的每一个外围设备子系统，它的使用是显而易见的。在这里可以设置 MCU 的所具有的中断、内存、定时器、IO 端口、UART、SPI 和模拟量比较器等外围设备，并产生相应的代码。如果需要的话，还可产生 main（ ）函数。

6. 状态窗口

状态窗口显示 IDE 的状态信息。

7. 终端仿真

IDE 有一个内置的终端仿真器，注意它不包含任意一个 ISP（在系统编程）功能，但它可以作为一个简单的终端，或许可以显示目标装置的调试信息，也可下载一个 ASC Ⅱ码文件。从 6. 20 版本开始，IDE 加入了对 ISP 的支持。

3. 6　菜单解释

1. 弹出菜单

在 ICCAVR 环境中单击右键，ICCAVR 会根据实际情况弹出相应的工具菜单。

2. File Menu（文件菜单）

New：新建一个文件，可在编辑窗口中输入文字或代码。

Reopen：重新打开历史文件，有关历史文件显示在右边的子菜单中。

Open：打开一个已经存在的文件用于编辑，文件用浏览窗口选择。

Reload. . . form Disk：放弃全部的修改，从磁盘中重新装载当前文件。

Reload. . . from Back Up：从最后一次的备份文件中装载当前文件。

Save：保存当前文件，如果环境设置中设置了保存备份文件，则将原文件以 <file>. ~<ext>形式保存。

Save as：将当前文件用另外一个名称来保存。

Close：关闭当前文件，如果文件有过修改，系统会进行提示。

Compile File... to Object：编译当前文件成目标文件，注意目标文件不可以直接用于对芯片编程或用于调试，其主要用于语法检查、为创建新的启动文件或库产生目标文件。

Compile File... to Output：编译当前文件成输出文件，其产生的输出文件可用于编程器和调试器。

Save All：保存所有打开的文件。

Close All：关闭当前打开的所有文件，同样它会提示保存已经修改的文件。

Print：打印当前文件。

Exit：退出 ICCAVR 的 IDE 环境。

3. Edit Menu（编辑菜单）

Undo：撤销最后一次的修改。

Redo：撤销最后一次的 Undo。

Cut：剪切选择的内容到剪贴板。

Copy：复制选择的内容到剪贴板。

Paste：将剪贴板内容粘贴在当前光标的位置。

Delete：删除选择的内容。

Select All：选择全部内容。

Block Indent：对选择的整块内容右移。

Block Outdent：对选择的整块内容左移。

4. Search Menu（寻找菜单）

Find...：在编辑窗口中寻找一个文本，它有以下选项：Match Case—区分大小写；Whole Word—全字匹配；Up/Down—往上或往下。

Find in Files...：在当前打开的文件中或在当前工程的所有文件中或当前目录中的文件中寻找一段文本，它有以下选项：Case Sensitive—大小写敏感；Whole Word—全字匹配；Regular Expression—寻找规则的表达式。

Replace...：在编辑器中替换文本。

Find Again：寻找下一个。

Goto Line Number：转到指定行号。

Add Bookmark：添加书签。

Delete Bookmark：删除书签。

Next Bookmark：跳转到下一个书签。

Goto Bookmark：跳转到指定的书签。

5. View Menu（视图菜单）

Status Window：如果选中，显示状态窗口。

Project Makefile：以只读方式打开 makefile 文件。

Output Listing File：以只读方式打开列表文件。

6. Project Menu（工程菜单）

New...：创建一个新的工程文件。

Open：打开一个已经存在的工程文件。

Open All Files...：打开工程的全部源文件。

Close All Files：关闭全部打开的文件。

Reopen...：重新打开一个最近打开过的工程文件。

Make Project：解释和编译已经修改的文件为输出文件。

Rebuild All：重新构筑全部文件，注意在版本升级后对原有工程最好全部重新构筑。

Add File（s）：添加一个文件到工程中，这个文件可以是非源文件。

Remove Selected Files：从工程中删除选择的文件。

Option...：打开工程编译选项对话框。

Close：关闭工程。

Save As...：将工程换一个名称存盘。

7. Tools Menu（工具菜单）

Environment Options：打开环境和终端仿真器选项对话框。

Editor and Print Options：打开编辑和打印选项对话框。

AVR Calc：打开AVR计算器，可以计算UART的波特率、定时器的定时常数。

Application Builder：打开应用向导程序，生成硬件的初始化代码。

Configure Tools：允许添加自己的内容到工具菜单。

Run：以命令行方式运行一个程序。

8. Compiler Options（编译选项）

编译选项总共有3个页面：Paths、Compiler和Target。

（1）Paths页面

Include Path（s）：可以指定包含文件的路径。

Assembler Include Path（s）：指定汇编包含文件的路径。

Library Path：链接器所使用的库文件的路径。

Output Directory：输出文件的目录。

在Compiler页面有：

Strict ANSI C Checking：严格的ANSI C语法检查。

Accept Extensions：接受C++类型语法扩充。

Macro Define（s）：定义宏，宏之间用空格或分号分开，宏定义形式如下：

name [：value]　或name [=value]

例如：DEBUG：1；PRINT=printf等价于#define DEBUG 1、#define PRINT printf

Macro Undefine（s）：同上，但意义相反。

Output File Format：输出文件格式COFF/HEX、Intel HEX或COFF。

Optimizations：代码优化。

Default：基本优化，像寄存器分配、共用相同的子例程等。

Maximize Code Size Reduction：只有专业版才可使用，它调用了代码压缩优化，去除了无用的碎片代码。

（2）Target页面

Device Configuration：选择目标 MCU。

Memory Sizes：要选择“Custom”时指定内存大小，包括 ROM、SRAM 和 EEPROM。

Text Address：通常代码地址开始于中断向量区域后面。

Data Address：指定数据起始地址，通常为 0x60。

Use Long JMP/CALL：指定 MCU 是否支持长跳转和长调用。

Enhanced Core：指定硬件支持增强核指令。

IO Registers Offset Internal SRAM：指定内部 SRAM 的偏移量。例如，8515 的 SRAM 起始于 0x60，　在 IO 寄存器空间后面延伸了 512B。而 Mega603，IO 寄存器覆盖在 SRAM 空间中，因此 SRAM 也是从 0 开始的。

Internal 对 External SRAM：指定目标系统的数据 SRAM 类型。

PRINTF Version：选择 PRINTF 的版本。

Small 或 Basic：只有%c、%d、%x、%X、%u 和%s 格式支持。

Long：支持%ld、%lu、%lx、%lX。

Floating point：%f 支持，注意这个选项需要很大的内存。

AVR Studio Simulator IO：如果选中，AVR Studio 的终端模拟仿真被支持。

Additional Libraries：使用标准库以外的附加库。

Strings in FLASH：字符串只保存在 FLASH 存储器中。

Return Stack Size：指定编译器使用的硬件堆栈的大小，编译器使用的软件堆栈的大小不需指定。

Non Default Startup：允许指定一个启动文件的位置，系统默认的启动文件在 Paths 页中指定，这样 IDE 可以使用多个启动文件。

Unused ROM Fill Pattern：用一串十六进制数填充空余的 ROM 空间。

3.7　C 库函数与启动文件

1. 启动文件

这个链接器会自动将启动文件连接到程序之前，并将标准库 libcavr.a 与程序相连接，启动文件根据目标 MCU 的不同在 crtavr.o 和 crtatmega.o 中任意选择一个。启动文件定义了一个全局符号_start，它也是程序的起点。启动文件的功能有：

1）初始化硬件和软件堆栈指针。

2）从 idata 区复制初始化数据到直接寻址数据区 data 区。

3）将 bss 区全部初始化为零。

4）调用用户主例程 main 函数。

5）定义一个退出点，如果主函数 main（ ）一旦退出，它将进入这个退出点进行无限循环。

启动文件也定义了复位向量，不需要修改启动文件来使用别的中断，具体可参考中断操作部分。

为修改和使用新的启动文件：

cd　\icc\libsrc.avr；　进入安装的编译器路径

<edit　crtavr.s>；　编辑修改 crtavr.s 文件

<open crtavr. s using the IDE>；　用IDE打开crtavr. s文件

<Choose" Compile File To－>Object">；　选择编译到目标文件,创建一个新的crtavr. o copy crtavr. o. . \lib；　复制到库目录

如果使用的目标MCU是Mega，应该用“crtatmega”代替“crtavr”，注意Mega的每个中断入口地址使用两个字，而非Mega芯片每一个中断入口地址使用一个字。

也可以有多个启动文件，可以在工程选项对话框中很方便地直接指定一个启动文件加入工程中。注意，必须指定启动文件的绝对路径或启动文件必须位于工程选项库路径所指定的目录中。

2. 常用库介绍

库源代码，这个库源代码（默认路径为c：\icc\libsrc. avr\libsrc. zip）是一个密码保护的ZIP压缩文件，可以从互联网上任意下载一个UNZIP程序进行解压缩。本软件被开锁后，密码显示在“About”对话框中，例如：unzip-s libsrc. zip；unzip　提示输入密码。

VR特殊函数，ICCAVR有许多访问UART、EEPROM和SPI的函数，堆栈检查函数对检测堆栈是否溢出很有用。另外互联网上有一个页专门存放用户写的源代码。

io*. h（io2313. h，io8515. h，iom603. h等），这些文件是从ATMEL官方公开的定义IO寄存器的源文件经过修改得到的，应该用这些文件来代替旧的avr. h文件。

PORTB=1；

uc=PORTA；

macros. h这个文件包含了许多有用的宏和定义。

还有其他头文件，下列标准的C头文件是被支持的，如果程序使用了头文件所列出的函数，那么包含头文件是一个好习惯，在使用浮点数和长整型数的程序中必须用#include预编译指令包含这些函数原形的头文件。读者可参考返回非整型值的函数。

assert. h：assert（），声明宏。

ctype. h：字符类型函数。

float. h：浮点数原形。

limits. h：数据类型的大小和范围。

math. h：浮点运算函数。

stdarg. h：变量参数表。

stddef. h：标准定义。

stdio. h：标准输入输出（IO）函数。

stdlib. h：包含内存分配函数的标准库。

string. h：字符串处理函数。

3. 字符类型库

下列函数按照输入的ACSⅡ字符集字符分类，使用这些函数之前应当用“#include <ctype. h>”包含。

int isalnum(int c)

如果c是数字或字母返回非零数值，否则返回零。

int isalpha(int c)

如果c是字母返回非零数值，否则返回零。

int iscntrl(int c)

如果c是控制字符（如FF、BELL、LF等）返回非零数值，否则返回零。

int isdigit(int c)

如果 c 是数字返回非零数值，否则返回零。

int isgraph(int c)

如果 c 是一个可打印字符而非空格返回非零数值，否则返回零。

int islower(int c)

如果 c 是小写字母返回非零数值，否则返回零。

int isprint(int c)

如果 c 是一个可打印字符返回非零数值，否则返回零。

int ispunct(int c)

如果 c 是一个可打印字符而不是空格、数字或字母返回非零数值，否则返回零。

int isspace(int c)

如果 c 是一个空格字符返回非零数值，包括空格 CR、FF、HT、NL 和 VT，否则返回零。

int isupper(int c)

如果 c 是大写字母返回非零数值，否则返回零。

int isxdigit(int c)

如果 c 是十六进制数字返回非零数值，否则返回零。

int tolower(int c)

如果 c 是大写字母则返回 c 对应的小写字母，其他类型仍然返回 c。

int toupper(int c)

如果 c 是小写字母则返回 c 对应的大写字母，其他类型仍然返回 c。

4. 浮点运算库

下列函数支持浮点数运算，使用这些函数之前必须用#include < math. h >包含。

float asin(float x)	以弧度形式返回 x 的反正弦值
float acos(float x)	以弧度形式返回 x 的反余弦值
float atan(float x)	以弧度形式返回 x 的反正切值
float atan2(float x,float y)	返回 y/x 的反正切,其范围在 $-\pi \sim +\pi$ 之间
float ceil(float x)	返回对应 x 的一个整型数,小数部分四舍五入
float cos(float x)	返回以弧度形式表示的 x 的余弦值
float cosh(float x)	返回 x 的双曲余弦函数值
float exp(float x)	返回以 e 为底的 x 的幂,即 e^x
float exp10(float x)	返回以 10 为底的幂,即 10^x
float fabs(float x)	返回 x 的绝对值
float floor(float x)	返回不大于 x 的最大整数
float fmod(float x,float y)	返回 x/y 的余数
float frexp(float x,int * pexp)	把浮点数 x 分解成数字部分 y(尾数)和以 2 为底的指数 n 两个部分,即 $x = y2^n$,y 的范围为 $0.5 \leq y \leq 1$,y 值被函数返回,而 n 值存放到 pexp 指向的变量中
float fround(float x)	返回最接近 x 的整型数
float ldexp(float x,int exp)	返回 $x \cdot 2^{exp}$
float log(float x)	返回 x 的自然对数

float log10(float x)　　返回以 10 为底的 x 的对数
float modf(float x,float * pint)把浮点数分解成整数部分和小数部分,整数部分存放到 pint 指向的变量,小数部分应当大于或等于 0 而小于 1,并且作为函数返回值返回
float pow(float x,float y)　　返回 x^y 值
float sqrt(float x)　　返回 x 的平方根
float sin(float x)　　返回以弧度形式表示的 x 的正弦值
float sinh(float x)　　返回 x 的双曲正弦函数值
float tan(float x)　　返回以弧度形式表示的 x 的正切值
float tanh(float x)　　返回 x 的双曲正切函数值

5. 标准输入输出库

标准的文件输入输出是不能真正植入微控制器（MCU）的，标准 stdio. h 的许多内容不可以使用，不过有一些 IO 函数是被支持的，同样使用之前应用“#include <stdio. h>”预处理，并且需要初始化输出端口，最底层的 IO 程序是单字符的输入（getchar）和输出（putchar）程序，如果针对不同的装置使用高层的 IO 函数，例如用 printf 输出 LCD，需要全部重新定义最底层的函数。

为了在 ATMEL 的 AVR Studio 模拟器（终端 IO 窗口）使用标准 IO 函数，应当在编译选项中选中相应的单选按钮。

注意：作为默认，单字符输出函数 putchar 是输出到 UART 装置没有修改，无论如何为使输出能如期望的那样出现在程序终端窗口中，‘\n’字符必须被映射为成对的回车和换行(CR/LF)。

int getchar()　使用查寻方式从 UART 返回一个字符

int printf(char * fmt,..)按照格式说明符输出格式化文本 frm 字符串,格式说明符是标准格式的一个子集

%d:输出有符号十进制整数;
%o:输出无符号八进制整数;
%x:输出无符号十六进制整数;
%X:除了大写字母使用‘A’~‘F’外,同%x;
%u:输出无符号十进制整数;
%s:输出一个以 C 中空字符 NULL 结束的字符串;
%c:以 ASCII 字符形式输出,只输出一个字符;
%f:以小数形式输出浮点数;
%S:输出在 FLASH 存储器中的字符串常量。

printf 支持 3 个版本，取决于特别需要和代码的大小（越高的要求，代码越大）：

基本型：　只有%c、%d、%x、%u 和%s 格式说明符是承认的；
长整型：　针对长整型数的修改%ld、%lu、%lx 被支持，以适用于精度要求较高的领域；
浮点型：　全部格式包括%f 被支持。

使用编译选项对话框来选择版本，代码大小的增加是值得关注的。

int putchar（int c）输出单个字符，这个库程序使用了 UART 以查寻方式输出单个字符，注意输出‘\n’字符至程序终端窗口。

int puts（char * s） 输出以 NL 结尾的字符串。

int sprintf（char * buf，char * fmt）按照格式说明符输出格式化文本 frm 字符串到一个缓冲区，格式说明符同 printf（ ）。

“const char * ” 支持功能，cprintf 和 csprintf 是将 FLASH 中的格式字符串分别以 prinf 和 sprintf 形式输出。

6. 标准库和内存分配函数

标准库头文件 <stdlib. h>定义了宏 NULL 和 RAND_ MAX 以及新定义的类型 size_ t，并且描述了下列函数。注意在调用任意内存分配程序（比如 calloc、malloc 和 realloc）之前，必须调用_ NewHeap 来初始化堆 heap。

int abs(int i)	返回 i 的绝对值。
int atoi(char * s)	转换字符串 s 为整型数并返回它,字符串 s 起始必须是整型数形式字符,否则返回 0。
double atof(const char * s)	转换字符串 s 为双精度浮点数并返回它,字符串 s 起始必须是浮点数形式字符串。
long atol(char * s)	转换字符串 s 为长整型数并返回它,字符串 s 起始必须是长整型数形式字符,否则返回 0。
void * calloc(size_t nelem,size_t size)	分配“nelem”个数据项的内存连续空间,每个数据项的大小为 size 字节并且初始化为 0。如果分配成功返回分配内存单元的首地址,否则返回 0。
void exit(status)	终止程序运行,典型的是无限循环,它是作为用户 main 函数的返回点。
void free(void * ptr)	释放 ptr 所指向的内存区。
void * malloc(size_t size)	分配 size 字节的存储区,如果分配成功则返回内存区地址。如内存不够分配则返回 0。
void_NewHeap(void * start,void * end)	初始化内存分配程序的堆,一个典型的调用是将符号_bss_end +1 的地址用作“start”值,符号_bss_end 定义为编译器用来存放全局变量和字符串的数据内存的结束,加 1 的目的是堆栈检查函数,使用_bss_end 字节存储为标志字节,这个结束值不能被放入堆栈中。

```
extern char_bss_end;
_New Heap(&_bss_end+1,&_bss_end+201);
```

初始化 200B 大小的堆

int rand(void)	返回一个在 0 和 RAND_MAX 之间的随机数。
void * realloc(void * ptr,size_t size)	重新分配 ptr 所指向内存区的大小为 size 字节,size 可比原来大或小,返回指向该内存区的地址指针。
void srand(unsigned seed)	初始化随后调用的随机数发生器的种子数。
long strtol(char * s,char * * endptr,int base)	按照“base.”的格式转换“s”中起始字符为长整型数,如果“endptr”不为空,* endptr 将设定“s”中转

换结束的位置。

unsigned long strtoul(char * s,char * * endptr,int base)

除了返回类型为无符号长整型数外,其余同“strtol”。

7. 字符串函数

用“#include <string. h>”预处理后，编译器支持下列函数。<string. h>定义了 NULL 类型 size_t 和下列字符串及字符阵列函数。

* memchr(void * s,int c,size_t n)

在字符串 s 中搜索 n 个字节长度寻找与 c 相同的字符，如果成功返回匹配字符的地址指针，否则返回 NULL。

int memcmp(void * s1,void * s2,size_t n)

对字符串 s1 和 s2 的前 n 个字符进行比较，如果相同则返回 0，如果 s1 中字符大于 s2 中字符则返回 1，如果 s1 中字符小于 s2 中字符，则返回 -1。

void * memcpy(void * s1,void * s2,size_t n)

复制 s2　中 n 个字符至 s1，但复制区不可以重叠。

void * memmove(void * s1,void * s2,size_t n)

复制 s2 中 n 个字符至 s1，返回 s1，其与 memcpy 基本相同，但复制区可以重叠。

void * memset(void * s,int c,size_t n)

在 s 中填充 n 个字节的 c，返回 s。

char * strcat(char * s1,char * s2)

复制 s2 到 s1 的结尾，返回 s1。

char * strchr (char * s, int c)

在 s1 中搜索第一个出现的 c，包括结束 NULL 字符。如果成功，返回指向匹配字符的指针，如果没有匹配字符找到，返回空指针。

int strcmp(char * s1,char * s2)

比较两个字符串，如果相同返回 0，如果 s1 >s2 则返回 1，如果 s1 <s2 则返回 -1。

char * strcpy(char * s1,char * s2)

复制字符串 s2 至字符串 s1，返回 s1。

size_t strcspn(char * s1,char * s2)

在字符串 s1 搜索与字符串 s2 匹配的第一个字符，包括结束 NULL 字符，其返回 s1 中找到的匹配字符的索引。

size_t strlen(char * s)

返回字符串 s 的长度，不包括结束 NULL 字符。

char * strncat(char * s1,char * s2,size_t n)

复制字符串 s2（不含结束 NULL 字符）中 n 个字符到 s1，如果 s2 长度比 n 小，则只复制 s2，返回 s1。

int strncmp(char * s1,char * s2,size_t n)

基本和 strcmp 函数相同，但其只比较前 n 个字符。

char * strncpy(char * s1,char * s2,size_t n)

基本和 strcpy 函数相同，但其只复制前 n 个字符。

char * strpbrk(char * s1,char * s2)

基本和 strcspn 函数相同但它返回的是在 s1 匹配字符的地址指针，否则返回 NULL 指针。

char * strrchr(char * s, int c)

在字符串 s 中搜索最后出现的 c，并返回它的指针。否则返回 NULL。

size_t strspn(char * s1, char * s2)

在字符串 s1 搜索与字符串 s2 不匹配的第一个字符，包括结束 NULL 字符，其返回 s1 中找到的第一个不匹配字符的索引。

char * strstr(char * s1, char * s2)

在字符串 s1 中找到与 s2 匹配的子字符串，如果成功，它返回 s1 中匹配子字符串的地址指针，否则返回 NULL。

"const char * "支持函数

这些函数除了它的操作对象是在 FLASH 中常数字符串外，其余同 c 中的函数。

size_t cstrlen(const char * s);

char * cstrcpy(char * dst, const char * src);

int cstrcmp(const char * s1, char * s2);

8. 变量参数函数

<stdarg. h>提供再入式函数的变量参数处理，它定义了不确定的类型 va_ list 和 3 个宏。

va_start(va_list foo, <last_arg>)	初始化变量 foo。
va_arg(va_list foo, <promoted type>)	访问下一个参数,分派指定的类型,注意那个类型必须是高级类型,如 int、long 或 double。小的整型类型如"char"不能被支持。
va_end(va_list foo)	结束变量参数处理。

例如，printf () 可以使用 vfprintf () 来实现。

```
#include <stdarg. h>
int printf( char * fmt,... )
{
va_list ap;
va_start( ap, fmt);
vfprintf( fmt, ap);
va_end( ap);
}
```

9. 堆栈检查函数

有几个库函数是用于检查堆栈是否溢出。如果硬件堆栈增长到软件堆栈中，那么软件堆栈的内容将会被改变。也就是说局部变量和别的堆栈项目被改变，硬件堆栈是用作函数的返回地址。如果函数调用层次太深，偶尔会发生这种情况。

同样地，软件堆栈溢出进数据区域将会改变全局变量或其他静态分配的项目（如果使用动态分配内存，还会改变堆项目)。这种情况在定义了太多的局部变量或一个局部集合变量太大时也会偶尔发生这种情况，如图 3-14 所示。

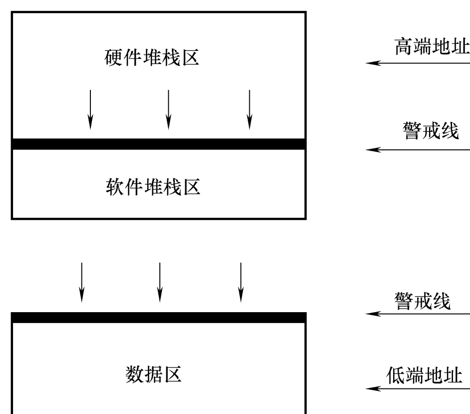


图 3-14　堆栈情况图

3.8　访问 AVR 单片机硬件的编程

1. 访问 AVR 单片机的低层硬件

AVR 系列单片机使用高级语言编程时有很高的 C 语言密度，它允许对访问目标 MCU 的底层硬件进行访问。由于 AVR 单片机性能，除了要最大程序地优化代码外很少使用汇编。偶然情况下目标 MCU 的硬件特点在 C 语言中不能很好地使用，很显然使用在线汇编和预处理宏能访问这些特点。

头文件 io *. h（如 io8515. h iom603. h 等）定义了指定 AVR 单片机 MCU 的 IO 寄存器细节。这些文件是从 ATMEL 官方发布的文件经过修改，以匹配这个编译器的语法要求。文件 macros. h 定义了许多有用的宏，例如宏 UART_TRANSMIT_ON（ ）能使 UART 开始工作。

这个编译器的效率很高，当访问由 IO 寄存器映射的内存时能产生单周期指令，如 in、out、sbis、sbi 等。参考 IO 寄存器。

注意：旧的头文件 avr. h 定义 IO 寄存器的 bit 有一些模糊，尽管 io *. h 定义了它们的 bit 的位置。因此使用 io *. h 和 IO 寄存器的 bit，很多时候将需要使用定义在 macros. h 文件中的 BIT（）宏。例如：

```
avr. h：
#define   SRE 0x80//   外部 RAM   使能
...   (C 程序)
MCUCR| =SRE;
io8515. h
#define SRE7
...   (C 程序)
#include   < macros. h >
MCUCR| =BIT(SRE);
```

2. 位操作

一个共同的任务是编程微控制器（MCU）打开或关闭 IO 寄存器的一些位（bit）。很幸运，标准 C 有较好的和适用的位操作功能，而没有借助于汇编指令或其他非标准 C 结构，C 定义了一些按位进行的运算是很有用的。

a | b：按位或，这个表达式指示中 a 被表达式中的 b 按位进行或运算。这惯用于打开某些位，尤其常用 |= 的形式，例如，PORTA |=0x80；//打开位 7（最高位）

a&b：按位与，这个运算在检查某些位是否置 1 时有用，例如，If((PORTA & 0x81) = = 0）//检查位 7 和位 0

注意，圆括号需要在 & 运算符的周围，因为它与 = = 相比运算优先级较低，这是 C 程序中很多错误的原因之一。

a^b：按位异或，这个运算对一个位取反有用，例如，在下面的例子中，位 7 是被翻转的：PORTA^ =0x80；//翻转位 7

~a：按位取反，在表达式中这个运算执行一个取反，当用按位与运算关闭某些位时，与这个运算组合使用尤其有用，如：PORTA & = ~0x80；//关闭位 7

这个编译器对这些运算能产生最理想的机器指令，例如，sbic 指令可以用在根据位的状

态进行条件分支的按位与运算中。

3. 程序存储器和常量数据

AVR 单片机是哈佛结构的 MCU，它的程序存储器和数据存储器是分开的，这样的设计是有一些优点的。例如，分开的地址空间允许 AVR 单片机装置比传统结构访问更多的存储器；例如，ATmega 系列允许有超过 64KW（字）的程序存储器和 64KB 的数据存储器。将来的 MCU 装置可能用到更多的程序存储器，而程序计数器仍保留在 16 位上。

不幸的是，C 不是在这种机器上发明的。特别地，C 指针是任意一个数据指针或函数指针，C 规则已经指定不可以假设数据和函数指针能被向前和向后修改，可是同是哈佛结构的 AVR 单片机，要求数据指针能指向任一个数据内存和程序内存。

非标准 C 解决了这个问题，ImageCraft AVR 编译器使用“const”限定词表示项目是在程序存储器中。注意对指针描述，这个 const 限定词可以应用于不同的场合，不管是限定指针变量自己还是指向项目的指针。例如：

```
const int table[ ] = {1,2,3};
const char * ptr1;
char *  const ptr2;
const char *  const ptr3;
```

“table”是表格式样分配在程序存储器，“ptr1”是一个项目在数据存储器而指向数据的指针在程序存储器，“ptr2”是一个项目在程序存储器而指向数据的指针在数据存储器，最后“ptr3”是项目在程序存储器而指向数据的指针也在程序存储器，在大多数的例子中“table”和“ptr1”是很典型的，C 编译器生成 LPM 指令来访问程序存储器。

注意，C 标准不要求“const”数据是放入只读存储器中，而且在传统结构中，除了正确访问就没有要紧的了。因而在承认参数的 C 标准中使用 const 限定是非传统的，无论如何这样做与标准 C 函数定义是有一定冲突的。

例如，标准“strcpy”的原型是 strcpy（char * dst，const char * src），带有 const 限定的第 2 个参数表示函数不能修改参数，然而在 ICCAVR 下，const 限定词表示第 2 个参数指向程序存储器是不合适的。因此这些函数定义设有 const 限制。

最后，注意只有常数变量以文件存储类型放入 FLASH 中，例如定义在函数体外的变量或有静态存储类型限制的变量，如果使用有 const 限制的局部变量，将不被放入 FLASH 中而可能导致不明确的结果。

4. 字符串

在哈佛结构的 AVR 单片机中，程序内存和数据内存分开，给程序内存和数据内存的说明带来了一定的复杂性，现在来讨论一下字符串。

这个编译器将带有 const 说明的表和项目放入程序存储器中。最困难的是字符串的分配和处理，问题在于 C 中将字符串转换为 char 指针。如果字符串是分配在程序存储器中，那么所有字符串库函数中的任意一个必须被复制成不同于指针的操作，或者字符串也必须被分配在数据存储器中。ImageCraft 编译器提出了解决这个问题的两个方法：

（1）默认的字符串

分配这个默认的方法是同时分配字符串在数据和程序存储器中，所有涉及的字符串是复制进数据存储器的，为了确保它们的值是正确的，在程序启动时字符串是由程序存储器复制进数据存储器中的。因此只有单一的字符串复制函数是必须的（编译器执行全局变量初始

化也是这样处理的)。

如果希望节省空间，能使用常量字符型数组来将字符串只分配进程序存储器中。例如：

const char hello[ ] = "Hello World";

在这个例子中，hello 可以在上下文中作为字符串使用，但不能用作标准 C 库中字符串函数的参数。

Printf 已被扩展成带%S 格式字符来输出只存储于 FLASH 中的字符串。另外，新的字符串函数已加入了对只存储于 FLASH 中字符串的支持。

(2) 只分配全部字符串到 FLASH 存储器中

当对应“Project-> Options-> Target-> Strings In FLASH Only”检查框被选中时，可以指挥编译器字符串只放在 FLASH 中，这时必须很小心地调用库函数。当这个选项是选中的，字符串类型“const char *”是有效的，并且必须保证函数获得了合适的参数类型。除了新的“const char *”与字符串有关系外，创建了 cprintf 和 csprintf 函数承认字符串格式的类型。读者可以参考标准输入输出函数。

注意，当选项 2（只分配全部字符串到 FLASH 存储器中）时，应使用 cprintf ( )。对 const char * 及 const char ptr [ ] 类型字符串，并且加%S 参数。

当选项 1 时，应使用 printf ( )。对 const char * 及 const char ptr [ ] 类型字符串，并且加%S 参数。

5. 堆栈

生成代码使用两个堆栈：一个是用于子程序调用和中断操作的硬件堆栈，另一个是用于以堆栈结构传递的参数、临时变量和局部变量的软件堆栈。

硬件堆栈起初是用于存储函数返回的地址，它代表了许多小的软件堆栈。通常，如果程序没有子程序调用，也不调用像带有%f 格式的 printf ( ) 等库函数。那么默认的 16B 应该在大多数的例子中能良好工作，在绝大多数程序中除了很繁重的递归调用程序（再入式函数)，最多 40B 的硬件堆栈应该是足够的。

硬件堆栈是从数据内存的顶部开始分配的，而软件堆栈是在它下面一定数量字节处分配。硬件堆栈和数据内存的大小是受在编译器选项中的目标装置项设定限制的，数据区从 0x60 开始分配，在 IO 空间后面是正确的，允许数据区和软件堆栈彼此相向生长。

如果选择的目标装置带有 32KB 或 64KB 的外部 SRAM，那么堆栈是放在内部 SRAM 的顶部，而且向低内存地址方向生长。可以参考程序和数据内存的使用。

任意一个程序失败的重要原因是堆栈溢出到其他数据内存的范围。两个堆栈中的任意一个都可能溢出，并且当一个堆栈溢出时会偶尔产生坏的事情，可以使用堆栈检查函数检测溢出情况。

6. 在线汇编

除了在汇编文件中写汇编函数外，在线汇编允许写汇编代码进 C 文件中（当然，在工程中使用汇编源文件作为一个部件是良好的)。在线汇编的语法如下：

asm(" <string> ");

多个汇编声明可以被符号\n 分隔成新的一行，“String”可以被用来指定多个声明，除了额外增加的 ASM 关键词。为了在汇编声明中访问一个 C 的变量，可使用% <变量名>格式，如：

register unsigned char uc;

```
asm("mov  %uc,R0\n""sleep\n");
```

任意一个 C 变量都可以被引用。如果在汇编指令中需使用一个 CPU 寄存器，必须使用寄存器存储类（register）来强制分配一个局部变量到 CPU 寄存器中。

通常，使用在线汇编引用局部寄存器的能力是有限的。如果在函数中描述了太多的寄存器变量，就很可能没有寄存器可用。在这种情况下，将从汇编程序得到一个错误，那时也不能控制寄存器变量的分配，所以在线汇编指令很可能失败。作为例子，使用 LDI 指令需要使用 R16～R31 中的一个寄存器，但这里没有请求使用在线汇编，同样也没有引用上半部分的整数寄存器。

在线汇编可以被用在 C 函数的内部或外部，编译器将在线汇编的每行都分解成可读的，不像 AVR 汇编器，ImageCraft 汇编器允许标签放置在任意地方，所以可以在在线汇编代码中创建标签。当汇编声明在函数外部时，可能得到一个警告，不要理睬这个警告。

7. IO 寄存器

IO 寄存器，包含状态寄存器 SREG，可以被两条路线访问。IO 地址在 0x00～0x3F 之间，可以使用 IN 和 OUT 指令读写 IO 寄存器；或者使用在 0x20～0x5F 之间的数据内存地址，可以使用普通数据访问指令和地址模式。两种方法在 C 中都可使用：

数据内存地址，一个直接地址可以通过加指针类型符号直接访问。例如，SREG 的数据内存在地址是 0x5F：

```
unsigned char c = *(volatile unsigned char *)0x5F;//  读 SREG
*(volatile unsigned char *)0x5F| =0x80;//  打开全局中断位
```

注意，数据内存地址 0～31 涉及 CPU 寄存器，注意不要改变 CPU 寄存器。

当访问在 IO 寄存器范围中的数据内存时，编译器自动生成低级指令如 in、out、sbrs、sbrc 等是首选的方法。

IO 地址，可以使用在线汇编和预处理宏来访问 IO 地址：

```
register unsigned char uc;
asm("in %uc,$3F");//  读 SREG
asm("out  $3F,%uc");//  打开全局中断位
```

注意，旧的头文件 avr. h 定义 IO 寄存器的 bit 有一些模糊，尽管 io *. h 定义了它们的 bit 的位置。因此在使用 io *. h 和 IO 寄存器的 bit 时，将需要使用定义在 macros. h 文件中的 BIT（）宏。例如：

```
avr. h:
#define  SRE 0x80//  外部 RAM  使能
...  （填充 C 程序）
MCUCR | =SRE;
io8515. h
#define  SRE 7
...  （填充 C 程序）
#include  <macros. h>
MCUCR| =BIT(SRE);
```

8. 绝对内存地址

程序可能需要使用绝对内存地址，例如外部 IO 设备通常被映射成特殊的内存。这些可

能包括 LCD 界面和双口 SRAM，通常可以使用在线汇编或单独的汇编文件来描述那些定位在特殊内存地址的数据。

在下面有例子中，假设有一个两字节的 LCD 控制寄存器定位在 0x1000 地址，一个两字节的 LCD 数据寄存器定位在 0x1002 地址，并且有一个 100 字节的双口 SRAM 定位在 0x2000 的地址。

使用汇编模式，在一个汇编文件中输入以下内容。

```
.area memory(abs)
.org 0x1000
_LCD_control_register::.blkw 1
_LCD_data_register::.blkw 1
org   0x2000
_dual_port_SRAM::.blkb 100
```

在 C 文件中必须这样描述：

```
extern unsigned int LCD_control_register,LCD_data_register;
extern char dual_port_SRAM[100];
```

注意，界面规定在汇编文件中外部变量名称是带‘_’前缀的并且使用两个冒号定义为全局变量。

也可以选择在线汇编，在线汇编遵守同样的汇编语法规则，除了它被附加了一个 asm() 伪函数。在 C 文件中，关于上面的汇编代码被变为如下代码：

```
asm(".area memory(abs)"
".org 0x1000"
"_LCD_control_register::.blkw 1"
"_LCD_data_register::.blkw 1");
asm(".org 0x2000"
"_dual_port_SRAM::.blkb 100");
```

在 C 中仍然要使用“extern”描述变量，正像上面使用单独的汇编文件那样，否则 C 编译器不会真正知道在 asm 中的声明。

9. C 任务（Tasks）

作为汇编界面的描述和调用规则，编译器通常生成代码来保存和恢复保护的寄存器。在一些情况下，这些行为可能是不合适的。例如，如果使用 RTOS（实时操作系统），RTOS 管理着寄存器的保存和恢复并作为任务切换处理的一部分，编译器如果再插入这些代码就变得多余了。

为了禁止这种行为，可以使用“#pragma ctask”，例如：

```
#pragma ctask drive_motor emit_siren
void drive_motor(){...}
void emit_siren(){...}
```

这个附注（pragma）必须被用在函数定义之前。注意作为默认的情况，从不返回的程序“main”是有这个属性的，它也没有必要为返回保存和恢复任意一个寄存器。

10. 中断操作

在中断操作中，C 中断可以使用，无论函数定义在文件的什么地方，必须用一个附注

（pragma）在函数定义之前通知编译器这个函数是一个中断操作：

#pragma interrupt_handler <name>：<vector number> *

“vector number”中断的向量号，注意向量号是从 1 开始的，那是复位向量。这个附注有两个作用：对中断操作函数，编译器生成 RETI 指令代替 RET 指令，而且保存和恢复在函数中用过的全部寄存器；编译器生成以向量号和目标 MCU 为基础的中断向量。

例如：

```
#pragma   interrupt_handler timer_handler:4
…
void   timer_handler( )
{
…
}
```

编译器生成的指令为

```
rjmp_timer_handler      ;对普通 AVR 单片机 MCU
```

或者

```
jmp_timer_handler       ;对 Mega 单片机 MCU
```

上述指令定位在 0x06（字节地址，针对普通装置）和 0x0c（字节地址，针对 Mega 装置）。Mega 使用 2 个字作为中断向量，非 Mega 使用 1 字作为中断向量。如果希望对多个中断入口使用同一个中断操作，可以在一个 interrupt_ handler 附注中放置多个用空格分开的名称，分别带有多个不同的向量号。例如：

```
#pragma interrupt_handler timer_ovf:7 timer_ovf:8
```

汇编中断操作，可以用汇编语言写中断操作。如果在汇编操作内部调用 C 函数，无论如何要小心。汇编程序要保存和恢复挥发寄存器（参考汇编界面），C 函数不做这些工作。

如果使用汇编中断操作，那么必须自己定义向量。使用“abs”属性描述绝对区域，用“. org”来声明 rjmp 或 jmp 指令的正确地址。注意这个“. org”声明使用的是字节地址。

```
;对全部除 ATmega 以外的 MCU
. area vectors(abs);中断向量
. org   0x6
rjmp_timer
;   对 ATmega MCU
. area   vectors(abs);中断向量
. org   0xC
jmp_timer
```

11. 访问 UART

默认的库函数 getchar 和 putchar 使用查寻模式从 UART 中进行读写。在 \icc\examples. avr 目录，有一个以中断方式工作的 IO 程序可以代替默认的程序。

12. 访问 EEPROM

EEPROM 在运行时可以使用库函数访问，在调用这些函数之前加入#include <eeprom. h>。

EEPROM_READ（int location，object），这个宏调用了 EEPROMReadBytes 函数，从 EEPROM 指定位置读取数据送给数据对象，“object”可以是任意程序变量包括结构和数组。例如：

```
int i;
EEPROM_Read(0x1,i);//  读 2B 给 i
```

EEPROM_WRITE（int location，object），这个宏调用了 EEPROMWriteBytes 函数，将数据对象写入到 EEPROM 的指定位置，“object”可以是任意程序变量包括结构和数组。例如：

```
int i;
EEPROM_WRITE(0x1,i);//写 2B 至 0x1
```

这些宏和函数可以用于任意 AVR 单片机装置。可是对 EEPROM 单元少于 256B 的 MCU，即使不需要高地址字节它们也是欠佳的，因为它仍然是要写的。如果它关系重大，可以为 EEPROM 较少的目标装置重新编译库源代码。

EEPROM 可以在程序源文件中初始化，在 C 源文件中它作为一个全局变量被分配到特殊调用区域“eeprom”中的，这是可以用附注实现的，结果是产生扩展名为 . eep 的输出文件。例如：

```
#pragma data:eeprom
int foo = 0x1234;
char table[ ] = {0,1,2,3,4,5};
#pragma data:data
...
int i;
EEPROM_READ((int)&foo,i);//i 等于 0x1234
```

第 2 个附注是必须的，为返回默认的“data”区域需要重设数据区名称。注意因为 AVR 单片机的硬件原因，初始化 EEPROM 数据至 0 地址是不可以使用的。注意当使用外部描述（比如访问在另一个文件中的 foo），不需要加入这个附注。例如：

```
extern int foo;
int i;
EEPROM_READ((int)&foo,i);
```

如果需要下列函数可以直接使用，上面关于宏的描述对大多数装置应该是有能力的。

unsigned char EEPROMread（int location），从 EEPROM 指定位置读取 1B。

int EEPROMwrite（int location，unsigned char byte），写 1B 到 EEPROM 指定位置，如果成功返回 0。

void EEPROMReadBytes（int location，void * ptr，int size），从 EEPROM 指定位置处开始读取“size”个字节至由“ptr”指向的缓冲区。

void EEPROMWriteBytes（int location，void * ptr，int size），从 EEPROM 指定位置处开始写“size”个字节，写的内容由“ptr”指向的缓冲区提供。

13. 访问 SPI

一个以查寻模式访问 SPI 的函数是提供的，更多的信息参考 spi. h。

14. 相对转移/调用的地址范围

一个带 8KB 程序存储器的装置，全部范围内的跳转可以使用相对转移和调用指令（rjmp 和 rcall）。为实现这个目的，相对转移和调用的范围是以 8KB 为分界的。例如，一个较远的跳转跳转到 0x2100 字节处（0x2000 为 8KB），实际上会跳转到地址 0x100 处。这个选项是由工程管理器自动检测的，只要目标装置的程序存储器是 8KB 的。

3.9　C 语言的运行结构

1. 数据类型（见表 3-1）

表 3-1　数据类型

类　　型	长度/B	范　　围
unsigned char	1	0 ~ 256
signed char	1	− 128 ~ 127
char①	1	0 ~ 256
unsigned short	2	0 ~ 65535
(signed) short	2	− 32768 ~ 32767
unsigned int	2	0 ~ 65535
(signed) int	2	− 32768 ~ 32767
unsigned long	4	0 ~ 4294967295
(signed) long	4	− 2147483648 ~ 2147483647
float	4	$+/-1.175e^{-38} \sim 3.40e^{+38}$
double	4	$+/-1.175e^{-38} \sim 3.40e^{+38}$

① char 等同于 unsigned char。

floats 和 doubles 是 IEEE 标准 32 位格式，7 位表示指数，23 位表示尾数，1 位表示符号。位域类型必须被赋予 unsigned 或 signed 关键字，而且将被包含在一个较小的空间中。如可定义成结构：

```
struct  {
unsigned a : 1, b : 1;
};
```

这个结构体的长度只有一个 1B。位域是从右往左放置的。

2. 汇编界面和调用规则

C 语言中的名称在汇编文件中是以下划线为前缀的，如函数 main（ ）在汇编模块中是以_main（ ）引用的。名称的有效长度为 32 个字符，在名称后面加两个冒号（::），可以定义成一个全局变量，例如：

```
_foo::
.word  1
(在 C  文件中)
extern  int  foo;
```

传递参数和返回值所使用的寄存器。第 1 个参数若是整型则通过 R16/R17 传递，第 2 个参数则通过 R18/R19 传递。如果参数是长整型或浮点数则通过 R16/R17/R18/R19 传递，其余参数通过软件堆栈传递。比如整型参数小的（如 char）参数扩展成整型（int）

长度传递，即使函数原型是可用的。如果 R16/R17 已传递了第 1 个参数，而第 2 个参数是长整型或浮点数，则第 2 个参数的低半部分通过 R18/R19 传递，而高半部分通过软件堆栈传递。

整型返回值是通过 R16/R17 返回，而长整型或浮点数返回则通过 R16/R17/R18/R19 返回。

在汇编函数中必须保护和恢复下列寄存器：R28/R29 或 Y，这是结构指针。R10/R11/R12/R13/R14/R15/R20/R21/R22/R23，这些寄存器是调用保护寄存器，这些寄存器的内容在被汇编语言函数调用后必须保持不变。

别的寄存器如 R0/R1/R2/R3/R4/R5/R6/R7/R8/R9/R24/R25/R26/R27/R30/R31、SREG 可以在汇编语言函数中使用，而不被保护和恢复。这些寄存器是调用挥发寄存器，这些寄存器的内容在被函数调用后可以改变。

对于它的中断处理，这不同于普通的函数调用，在中断操作中必须保护和恢复它所使用的全部寄存器。如果使用 C 函数来描述中断处理，那么编译器有能力自动完成。如果使用汇编写中断处理，而它又调用了普通的 C 函数，那么汇编操作必须保护和恢复挥发性寄存器，普通 C 函数调用不保护它们。中断处理操作同普通程序操作是异步的，中断处理或它的函数调用不能改变任意一个 MCU 寄存器。

3. 函数返回非整型值

在调用函数前，必须描述其返回的一个长整型、浮点数或结构值。作为例子，在调用任意浮点函数之前，应当用#include 语句包含头文件 <math. h>。否则，在这些程序返回它们的值后程序将不工作，这与那些返回整型值的函数是有不同之处的。

长整型数或浮点数返回值设定在寄存器 R16 ~ R19 中。如果传递结构值的话，结构允许通过堆栈传递，而不是通过寄存器。传递结构索引（也就是传递结构的地址）和传递任意数据项目的地址是相同的，都是通过一个 2B 的指针。对于返回结构值，当一个返回结构的函数被调用时，这个调用函数分配一个临时存储库，而且传递一个隐藏指针给调用函数。当这个函数返回时，它复制返回值到这个临时存储库。

4. 程序和数据区的使用

程序存储器是被用于保存程序代码、常数表和确定数据的初始值，比如字符串、全局变量。编译器可以生成一个对应程序存储器映像的输出文件（HEX 文件），这个文件可以被编程器用来对芯片编程。

通常，编译器不能使用任意 64KB 以上的程序存储器。为了访问 64KB 边界以上的存储器（如在 Mega103 装置中），要在设定 RAMPZ 寄存器后直接调用 ELPM 指令。

数据存储器（仅指内部 SRAM），这个数据存储器是被用于保存变量、堆栈结构和动态内存分配的堆。通常，它们不产生输出文件，但在程序运行时使用，一个程序使用数据内存如图 3-15 所示。

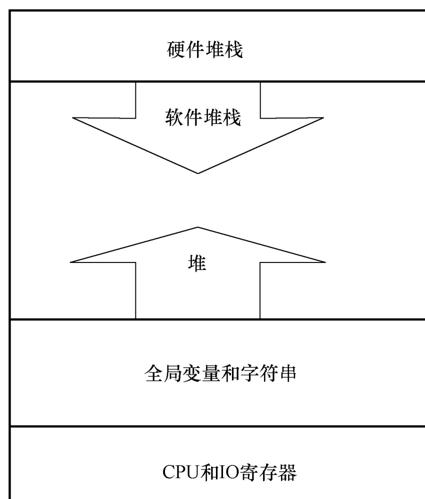


图 3-15　数据存储器

3.10　其他主流 AVR 单片机开发环境简介

3.10.1　GCCAVR 开发环境

GCCAVR 是著名的自由软件编译器 GNU GCC 的 AVR 平台的移植。其包括两部分：编译和链接的命令行程序包以及针对 AVR Libc 的函数库。如同其他 GNU 协议下的软件一样，所有这些都是以源程序形式发布的，用户可以根据其自身的计算机平台进行配置、编译，生成适合用户自身计算机平台的可执行的 GNU GCC AVR 版本。对于 Windows 用户，也有已经预先编译好的二进制版本可供下载。

GNU GCC AVR 的特点如下：

1）所有源代码都是向用户开发的，完全免费。

2）GNU GCC AVR 支持 ANSI C/C + +/EC + +；

3）GNU GCC AVR 编译后的代码执行效率和稳定性仅次于 IAR Systems 的 Embedded Workbench 编译器。

4）支持几乎所有的 AVR 单片机器件；

5）包括兼容 ANSI C 的部分标准函数库和针对 AVR 单片机的各个外设的函数库；

6）缺乏专业的技术支持，缺乏图形的集成开发环境（IDE），所有程序都由命令行执行。

3.10.2　CodeVision AVR 集成开发环境

CodeVision AVR 是 HP Info Tech 专门为 AVR 单片机设计的一款低成本的 C 语言编译器。它产生的代码非常严密，效率很高。它不仅包括了 AVR C 编译器，同时也是一个集成 IDE 的 AVR 单片机开发平台，简称 CVAVR。

CVAVR 支持所有片内含有 RAM 的 AVR 芯片，具备以下特点：

1）支持 bit、char、short、long、float 以及指针等多种数据类型，并充分利用存储器：

2）内建有针对 AVR 单片机优化的多种选项；

3）支持内嵌汇编；

4）对一些标准外部器件的支持和接口函数（如标准字符 LCD 显示器、I^2C 接口、SPI 接口、延时、BCD 码与格雷码转换等）；

5）可以直接在 C/C + +中编写快速、易用的中断处理函数；

6）高效的 32 位和 64 位的 IEEE 兼容的浮点运算；

7）扩展的 C 和 EC + +的函数库，并可进行数学和浮点运算。

3.10.3　IAR 集成开发环境

IAR Systems 的 Embedded Workbench 集成了一个集成环境，包括编译器和图形化的调试工具，能够完成系统的设计、测试和文档工作，还可以同时打开多个项目，以及很容易扩展集成诸如代码分析等外部工具。

其 C 编译器和汇编编译器支持几乎所有的 AVR 单片机芯片，具备以下特点：

1）C 编译器支持 ISO/ANSI C 的标准 C 和可选的 Embedded C + +编译器；

2）所有代码都可重入；
3）有多种存储器模型和指针类型，以充分利用存储器；
4）内建有针对 AVR 单片机优化的选项、多重的代码大小和执行速度的优化控制；
5）针对 AVR 单片机的语言扩展，以适应嵌入式编程；
6）新增的强大全局优化器；
7）可以直接在 C/C + + 中编写快速、易用的中断处理函数；
8）高效的 32 位和 64 位的 IEEE 兼容的浮点运算；
9）扩展的 C 和 EC + + 的函数库，并可进行数学和浮点运算。

第 4 章　C 语言概论、数据类型、运算符与表达式

4.1　C 语言概论

4.1.1　C 语言的发展过程

C 语言问世于 1969～1973 年间，1978 年由美国电话电报公司（AT&T）贝尔实验室正式发表了 C 语言，同时 B. W. Kernighan 和 D. M. Ritchit 两人一起撰写了著名的《THE C PROGRAMMING LANGUAGE》一书。通常简称为《K&R》或“白皮书”。但是，在“白皮书”中并没有对 C 语言标准有一个完整的定义，直到 1983 年，C 语言标准才由美国国家标准学会在“白皮书”的基础上制定，我们将其称之为 ANSI C。到了今天，C 语言已经成为一门普遍应用于计算机行业中的语言了。

4.1.2　C 语言的特点

C 语言是一种结构化语言。首先，它层次清晰，便于按模块化方式组织程序，易于调试和维护，语言紧凑，使用方便、灵活。其次，它具有丰富的运算符和数据类型，便于实现各类复杂的数据结构。第三，可以直接访问内存地址，能进行位（bit）操作的特点，使其能够胜任开发操作系统的工作。第四，由于 C 语言可以对硬件进行编程操作，因此它既有高级语言的功能，也有低级语言的优势。不仅可用于系统软件的开发，同时也适用于应用软件的开发。另外，C 语言还具有效率高，可移植性强等特点。例如，原来使用汇编语言编写的程序，由于别人写的程序不易被读懂，在一段时间后再去做升级和维护就会感觉非常不便，但在使用和维护 C 语言写的程序时，就不会遇到这样的困扰，这时 C 语言的优势就大大体现出来了。

4.1.3　C 源程序的结构特点

用下面两个典型的程序例子来说明 C 语言在组成结构上的特点。同时，也可以从这两个由简到难的程序例子中，了解到组成一个 C 源程序的基本部分和书写格式。

【例 4.1】

```
main()
{
printf("朋友,您好！ \n");
}
```

main 是主函数的函数名，表示这是一个主函数。每一个 C 程序都必须有，而且只能有一个主函数（main 函数）。函数调用语句 printf 函数的功能是在显示器上显示要输出的内容。printf 函数是一个由系统定义好的标准函数，可在程序中直接调用。

【例 4.2】

```
int min(int a,int b); /* 函数声明 */
```

```
main(){ /* 主函数 */
int x,y,z; /* 变量声明 */
printf("input two numbers:\n");scanf("%d%d",&x,&y); /* 输入 x,y 值 */
z = min(x,y); /* 调用 min 函数 */
printf("min num = %d",z); /* 输出 */
}
int min(int a,int b){ /* 定义 min 函数 */
if(a < b)return a;else return b; /* 把结果返回主函数 */
}
```

此函数的功能是由用户输入两个整数，然后输出其中一个较小的数。

例 4.2 程序的功能是由用户输入两个整数，程序执行后输出其中较小的数。本程序由两个函数组成，一个主函数，一个 min 函数。函数之间是并列的关系。在主函数中可以调用其他函数。min 函数的功能是比较两个数，然后把较小的数返回给主函数。min 函数是一个用户自定义函数。因此在主函数中要给出声明（程序的第一行）。由此可见，在程序的声明部分中，不仅可以有变量的声明，还可以有函数的声明。关于函数的详细内容将在第 5 章介绍。在例程中，可以看到程序每行后面有用/*和*/括起来的内容，称之为注释部分，程序在编译时，不会执行这部分内容。

例 4.2 程序执行的过程是，首先出现让用户输入两个数的提示信息，scanf 函数的作用是将这两个数送入变量 x、y 中，然后调用 min 函数，并把 x、y 的值传送给 min 函数的形式参数 a、b。min 函数中的语句用来比较变量 a、b 数值的大小，把小的那个数返回给主函数的变量 z，最后再显示输出 z 的值。

C 语言的特点如下：

1）一个 C 语言源程序可以由一个或多个源文件组成。

2）每个源文件可由一个或多个函数组成。

3）一个源程序不论由多少个文件组成，都有一个且只能有一个 main 函数，即主函数。

4）源程序中可以有预处理命令（include 命令仅为其中的一种），预处理命令通常应放在源文件或源程序的最前面。

5）每一个声明、每一个语句都必须以分号结尾。但预处理命令、函数头和花括号“}”之后不能加分号。

6）标识符、关键字之间必须至少加一个空格以示间隔。若已有明显的间隔符，也可不再加空格来间隔。

4.1.4　C 语言的字符集

字符是组成语言的最基本的元素。字母、数字、空格、标点和特殊字符组成 C 语言字符集。在字符常量、字符串常量和注释中还可以使用汉字或其他可表示的图形符号。

1）字母：小写字母 a ~ z 共 26 个，大写字母 A ~ Z 共 26 个。

2）数字：0 ~ 9 共 10 个。

3）空白符：空格符、制表符、换行符等统称为空白符。空白符只在字符常量和字符串常量中起作用。在其他地方出现时，只起间隔作用。编译程序对它们忽略不计。因此在程序中是否使用空白符，不影响程序的编译，但在程序中适当的地方使用空白符，可以增加程序

的清晰性和可读性。

4）标点和特殊字符。

4.1.5　C 语言的词汇

在 C 语言中使用的词汇分为 6 类：标识符、关键字、运算符、分隔符、常量、注释符等。

1. 标识符

在程序中使用的变量名、函数名、标号等统称为标识符。除库函数的函数名由系统定义外，其余都由用户自定义。C 语言规定，标识符只能是字母（A～Z，a～z）、数字（0～9）、下划线（_ ）组成的字符串，并且其第一个字符必须是字母或下划线。

以下标识符是合法的：a，BOOKS ，abc5。

以下标识符是非法的：4s，以数字开头；s#T，出现非法字符#。

在使用标识符时还应注意以下几点：

1）由于 C 语言编译器版本不同，导致标识符的长度也会有所不同。例如有些版本 C 语言中规定标识符前 8 位有效。

2）在标识符中，大小写是有严格区分的，例如 TEACHER 和 teacher 就是两个不同的标识符。

3）定义标识符时，取的名字应尽量有直观的意义，以便于阅读理解，做到“顾名思义”。

2. 关键字

关键字是在 C 语言系统中具有特定意义的字符串，通常也称为保留字。用户定义的标识符名字不能与关键字相同。C 语言的关键字分为以下几类：

（1）类型声明符

用来定义变量、函数或其他数据结构的类型。如例程中用到的 int、double 等。

（2）语句定义符

用来表示一个语句的功能意义。如后面要讲到的“if else”就是条件语句的语句定义符。

（3）预处理命令字

表示预处理命令的关键字。如例程中用到的“include”。

3. 运算符

C 语言中含有很丰富的运算符。运算符是告诉编译程序执行特定算术或逻辑操作的符号。C 语言有 3 大运算符：算术、关系与逻辑、位操作。另外，C 语言还有一些用于完成特殊任务的特殊运算符。

4. 分隔符

C 语言中的分隔符有逗号和空格两种。逗号主要用于数据类型声明和函数参数表中，分隔各个变量；而空格多用于语句各单词之间，做间隔符。

5. 常量

C 语言中使用的常量可分为数字常量、字符常量、字符串常量、符号常量、转义字符等几种。

6. 注释符

C 语言的注释符是从以“/＊”开头到以“＊/”结尾的内容，在“/＊”和“＊/”之间的内容即为注释。在编译程序时，不对这些注释内容做任何处理。注释可出现在程序中的

任何位置，起到向用户提示或解释程序意义的作用，同时也为编写及调试、维护程序工作提供了便利。

4.2　数据类型、运算符与表达式

4.2.1　C 语言的数据类型

计算机中的程序离不开数据这个单元，它是计算机操作的对象，数据的不同格式叫做数据类型；而数据结构则是按一定的数据类型进行一些组合和构架。

在 C 语言中，数据类型分为基本类型、构造类型、指针类型、空类型 4 大类，如图 4-1 所示。

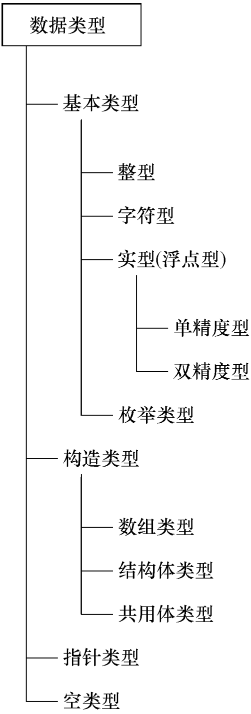


图 4-1　数据类型

4 种数据类型的特点：

1）基本类型：它的数据不可以再进行分解。

2）构造类型：根据已定义的一个或多个数据类型用构造的方法来定义的。也就是说，一个构造类型的值可以由若干个“成员”或“元素”组成。其“成员”可以是一个基本数据类型或构造类型。在 C 语言中，构造类型有数组类型、结构体类型、共用体类型等几种。

3）指针类型：指针是一个特殊的变量，它里面存储的数值被解释成为内存里的一个地址，也是 C 语言的精华所在，虽然指针变量的值类似于整型量，但从其数据类型意义来看，这完全是两种完全不同的类型，所以不能混为一谈。

4）空类型：函数在被调用完成后，通常会返回一个函数值。函数值有一定的数据类型的，应在函数定义及函数声明中加以声明，例如在例 4.2 中给出的 min 函数定义中，函数头为 int min（int a，int b）；其中“int ”表示 min 函数的返回值为整型数据类型。但是，也经常会碰到一些函数，调用后并不需要向调用者返回函数值，这时如何来书写呢？此时，将这种函数定义为“空类型”，其类型声明符为 void。如将 int min（int a，int b）；改为 void min（int a，int b）；即可。

在本章中，先介绍基本数据类型中的整型、浮点型和字符型。其余几种类型将在以后各章中陆续介绍。

1. 常量与变量

对于基本数据类型量，根据变量值在程序执行过程中是否发生变化，又可分为常量和变量两种。在程序执行过程中，其值不发生改变的量称为常量，其值可变的量称为变量。每个变量都会有名字，并在内存中占据一定的存储单元，所占存储单元的数量根据数据类型的不同而不同。在程序中，常量是可以不经声明直接引用的，而变量则必须先定义后使用。

（1）常量和符号常量

常量与变量相对应，在程序执行的过程中，其值不能发生改变的量称为常量。常量与变量不同，可以有不同的数据类型，如 1、3、5 为整型常量，6.9、－1.85 为实型常量，‘c’，‘d’为字符常量。常量可以用一个标识符来声明。

符号常量在使用之前必须先定义，其一般形式为：

#define 标识符 常量

其中#define 是一条编译预处理命令（预处理命令都以" 0#" 开头），称为宏定义命令，它的功能是把该标识符定义为其后的常量值。定义之后，凡在程序中所有出现该标识符的地方均用之前定义好的常量来代替，习惯上用大写字母来表示符号常量的标识符，用小写字母表示变量的标识符，以示区别。

【例 4.3】

```
#define PI 3.14
main()
{
float r,s;
r=5.3;
s=PI*r*r;
printf("半径为 R 的圆面积为%f",s);
}
```

程序的第一句话“#define PI 3.14”定义了一个符号常量 PI，它的值为圆周率 3.14，由此一来，在后面的程序代码中，出现 PI 的地方，都代表 3.14 这个数。

使用符号常量的优点是：意义清楚，修改参数非常方便，如当程序中很多地方都要用到这个变量，而数值又需要经常做改动，这时使用符号常量就可以做到“一改全改，一次改成”。

（2）变量

在程序运行中，其值可以改变的量称为变量。一个变量只有一个名字，在内存中占据一定的存储单元，如图 4-2 所示。变量必须先定义再使用，一般放在函数体的开头部分，全局变量放在函数体外面。

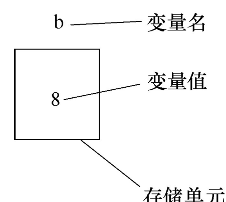


图 4-2　变量与存储单元

2. 整型数据

（1）整型常量

整型常量就是整型的常数。在 C 语言中，整型常量可分为八进制、十进制和十六进制 3 种。

1）十进制数没有前缀。用数码 0～9 来表示。

如以下各数就是合法的十进制整型常数：257、－518、55535、1968。

在程序中是根据前缀来区分各种进制数的。因此在书写常数时不要把前缀弄错导致结果不正确。

2）八进制数：必须以 O 开头，即以 O 作为八进制数的前缀。用数码 0～7 来表示。八进制数通常是无符号数。

以下各数是合法的八进制数：O16（相当于十进制数 14）、O110（相当于十进制数 72）、O177776（相当于十进制数 65534）；

以下各数不是合法的八进制数：156（无前缀 O）、O1B2（包含了非八进制数字）、－O170（不应该有负号）。

3）十六进制数：以 OX 或 Ox 开头。其数码取值为 0～9、A～F 或 a～f。

以下各数是合法的十六进制整常数：OX2B（相当于十进制 43）、OXB0（相当于十进制 176）、OXFFFF（相当于十进制 65535）；

以下各数不是合法的十六进制整常数：7B（无前缀 OX）、OX5H（含有非十六进制数码）。

（2）整型变量

整型变量的分类

1）基本型：类型声明符为 int，在内存中占 2B。

2）无符号型：类型声明符为 unsigned 。无符号型又可为 unsigned int 或 unsigned。无符号类型量所占的内存空间字节数与相应的有符号类型量相同，但由于省去了符号位，故不能表示负数，也因此表示正数的数值范围有所扩大。

有符号整型变量：最大表示 32767

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

无符号整型变量：最大表示 65535

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

整型变量的定义

变量定义的一般形式为：

类型声明符　变量名标识符，变量名标识符，...；

例如：

int a，b，c；(a、b、c 为整型变量)

unsigned int x，y；(x、y 为无符号整型变量)

定义变量时，允许在一个类型声明符后，同时定义多个相同类型的变量。这时，各变量名之间用逗号间隔，类型声明符与变量名之间至少用一个空格间隔。

【例 4.4】 整型变量的定义与使用。

```
main( )
{
int a,b;
unsigned c;
a = 1;b = 4;
c = a + b;
printf("a = %d,b = %d,c = %d\n",a,b,c);
}
```

3. 实型数据

（1）实型常量

在 C 语言中，实型数也称为浮点型。它有 2 种表示形式：十进制小数形式和指数形式。

1）十进制数形式：由数码 0 ~ 9 和小数点组成。

例如：0.0、24.0、5.189、0.93、90.0、6000.、-227.8670 等均为合法的实数。在这个数中是必须存在小数点的。

2）指数形式：在十进制数的基础上，加阶码标志“e”或“E”和阶码（只能为整数，可以带正负号）组成。

其一般形式为：a E n（a 为十进制数，n 为十进制整数）

表示的数值为 $a \times 10^n$。

如：

3. 1E4（等于 3.1×10^4）

1. 6E-2（等于 1.6×10^{-2}）

0. 8E7（等于 0.8×10^7）

－1. 14E-2（等于 -1.14×10^{-2}）

以下不是合法的实数：

385（无小数点）

E6（阶码标志 E 之前无数字）

－1（无阶码标志）

51. －E3（负号位置不对）

2. 9E（无阶码）

标准 C 语言允许浮点数尾部加后缀，“f”或“F”即表示该数为浮点数。如 328f 和 328. 是等价的。

【例 4. 5】 通过以下程序例子可以看到其显示输出值都是一样的。

```
main ( ){
      printf( "%f\n" ,356. ) ;
      printf( "%f\n" ,356) ;
      printf( "%f\n" ,356f) ;
}
```

（2）实型变量

实型变量主要分为单精度（float 型）和双精度（double 型），见表 4-1。

在 C 编译器中单精度型占 4B（32 位）内存空间，其数值范围为 3. 4E-38 ~ 3. 4E + 38，只能提供 7 位有效数字。双精度型占 8B（64 位）内存空间，其数值范围为 1. 7E-308 ~ 1. 7E + 308，提供 16 位有效数字。

表 4-1　float 与 double 类型数据

类型声明符	比特数(字节数)	有效数字	数的范围
float	32(4)	6 ~ 7	$10^{-38} \sim 10^{38}$
double	64(8)	15 ~ 16	$10^{-308} \sim 10^{308}$

实型变量定义的方法与整型相同，例如：

float x，y；（x、y 为单精度实型量）

double a，b，c；（a、b、c 为双精度实型量）

4. 字符型数据

字符型数据分为字符常量和字符变量。

（1）字符常量

用单引号括起来的一个字符，称为字符常量。

例如：‘y’、‘f’、‘－’、‘＋’、‘/’这些都是合法字符常量。

在使用字符常量时，需要注意下面几点：

1）字符常量只能用单引号括起来，用双引号或其他括号，将出现错误提示。

2）字符常量只能是单个字符，不能出现多个字符。

3）字符可以是字符集中的任意字符。但如果将数字定义为字符型常量后，它就不能参加数值运算了。如‘8’和 8，仔细看一下是不同的。前者是字符常量，不能参与运算，而后者才是一个整型数据。

（2）转义字符

在 C 语言中，转义字符是一种特殊的字符常量。它是以反斜线“ \ ”开头的字符序列。不同的转义字符具有不同的特定含义，因此称它为“转义”字符。例如，经常在例题中看到的 printf 函数中用到的“ \ n”就是一个转义字符，其意义是“回车换行”。转义字符主要用来表示那些用一般字符不便于表示的语句代码。表 4-2 列出了 C 语言常用的转义字符及含义。

表 4-2 C 语言常用的转义字符及含义

转义字符	转义字符的意义	ASCII 代码
\n	回车换行	10
\t	横向跳格(跳到下一输出区)	9
\b	退格	8
\r	回车	13
\f	走纸换页	12
\\	反斜线符“\”	92
\’	单引号符	39
\”	双引号符	34
\a	报警	7
\ddd	1 ~3 位八进制数所代表的字符	
\xhh	1 ~2 位十六进制数所代表的字符	

可以说，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的 \ ddd 和 \ xhh 正是起到了这个作用。ddd 和 hh 分别为八进制 ASCII 码和十六进制 ASCII 码。如 \ 102 表示字母“B”，\ 103 表示字母“C”，\ XOA 表示换行等。

【例 4.6】 转义字符的使用方法。

```
main()
  {
  int a,b,c;
  x =5; y =6; z =7;
  printf("   xy   z\tde\rf\n");
  printf("hijk\tL\bM\n");
  }
```

大家可以想一下，这个程序的输出结果是怎么样的。

（3）字符变量

字符变量用来存储单个字符。

字符变量的类型声明符是 char。其定义方法与上面所讲的数据类型定义方法一样，例如：char a，b；

char 数据类型的长度是 1B，通常用于定义处理字符数据的变量或常量。unsigned char 与 signed char 数据类型的区别是有无符号位，如果直接使用 char 类型的话，其默认值为 signed char 类型。因为 unsigned char 类型 1B 所有的 8 位均来表示数值，而 signed char 类型用字节中的最高位来表示数据的正负，“0”表示其为正数，“1”表示为负数，所以 unsigned char 类型

可以表达的数值范围是 0 ~255，而 signed char 类型可以表达的数值范围是 -128 ~ +127。

（4）字符串常量

字符串常量是由一对双引号括起的字符序列。例如："CHINA"、"C program"、"$ 1.3"等都是合法的字符串常量。

字符串常量与字符常量之间的不同之处总结为以下几点：

1）字符常量使用单引号括起来，而字符串常量使用双引号括起来。

2）字符常量只能是单个字符，而字符串常量却可以是一个或多个字符。

3）可以把一个字符常量赋给一个字符变量，但反过来把一个字符串常量赋予一个字符变量是不行的。在 C 语言中是没有相应的字符串变量的。

4）字符常量在内存中占 1B 的空间。字符串常量在内存中所占字节的大小等于字符串的字节数加 1，这是一个字符串结束的标志位，在 C 语言中，用字符'\0'作为字符串的结束标志，'\0'的 ASCII 码值为 0。

例如，字符串"C program"在内存中所占字节的情况为：

C		p	r	o	g	r	a	m	\0

'\0'为字符串"C program"结束的标志位。

字符常量'b'和字符串常量"b"，从表面上来看，虽然都只有一个字符，但在内存中的存储情况却是不同的。

'b'在内存中占 1B，其存储情况为：

b

"b"在内存中占 2B，其存储情况为：

b	\0

5. 变量赋初值

在程序中常常需要对变量赋予一个初始值。C 语言可以在做变量定义的同时，给变量赋初值，也称之为变量的初始化操作；也可以在后面的语句中再给变量赋值。

变量初始化赋初值的一般形式为：

类型声明符 变量 1 = 值 1，变量 2 = 值 2，变量 3 = 值 3……；

例如：

```
int b =8;
int c,e =1;
double a =8.2,b =6f,c =0.575;
char ch1 ='A',ch2 ='B';
```

与其他高级编程语言不同，C 语言在变量定义中不允许出现多个"="赋值符号，如 a = b = c =3 是非法的。

【例 4.7】

```
main()
{
 int aa =3,bb,cc =5;
 bb =aa +cc;
```

```
  printf("aa = %d,bb = %d,cc = %d\n",aa,bb,cc);
 }
```

6. 各类数值型数据之间的混合运算

变量的数据类型是可以转换的。转换方式有两种，一种是自动类型转换，另一种是强制类型转换。当程序中不同数据类型的变量混合运算时，自动类型转换由编译器自动完成。自动类型转换遵循以下转换规则：

1）若参与运算的数据类型不一样，则先转换成同一数据类型，再进行运算。

2）转换以保证精度不降低的原则进行。

3）所有的浮点运算都是转换成双精度进行的，即使表达式中仅含 float 型单精度的数据运算量，也要先转换成 double 型，再作运算处理。

4）在赋值运算中，当赋值号两边的数据类型不同时，赋值号右边的数据类型将转换成左边的数据类型。如果右边的数据类型长度大于左边的数据类型长度时，那它将丢失一部分数据，丢失的部分遵循按四舍五入的原则，降低了精度。

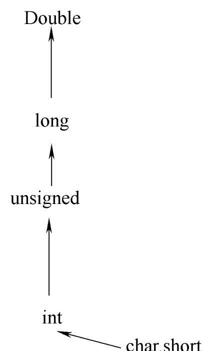


图 4-3　不同数据之间的自动类型转换

图 4-3 所示为自动类型转换的规则。

当 int 与 char 型同时进行运算时，则系统先将 char 转为 int 型数据；当 float 与 double 型共同存在时，则先将 float 型转为 double 型数据。当 int、unsigned、long、double 类型的数据同时存在时，则其转换高低关系为 int- > unsigned- > long- > double。如果 int 与 long 型共存时，则必将 int 型转换为 long 类型数据。

【例 4.8】

```
main()
{
float PI =3.1415;
int s;
int r =6;
s =r * r * PI;
printf("面积:s = %d\n",s);
}
```

例 4.8 中变量 PI 为实型；s，r 为整型。当程序运行到 s = r * r * PI 语句时，变量 r 和变量 PI 都将转换成 double 型数据，其计算结果也为 double 类型，但因为 s 为整型变量，根据自动类型转换原则，其最终结果应该为整型，如出现小数数值，则舍去了小数部分，所以程序的执行结果其面积 s = 113。

强制类型转换是通过类型转换运算符来实现的。

一般形式为：

（类型名）　（表达式）

它的功能是把表达式的运算结果值强制转换成类型名所表示的数据类型。

例如：

(double) a　将 a 转换为 double 类型数据

(int) (x + y)　将 x + y 的值转换成整型

（float）（5%3）将 5%3 的值转换成 float 型

在使用强制转换时应注意以下两点：

1）当变量为多个时，类型名和表达式都必须加上括号（单个变量可以不加括号），如把（int）（x + y）写成（int）x + y，则意义为把 x 转换成 int 型后再与 y 相加；而（int）（x + y）的意义是把 x 和 y 相加，其相加结果再转换成 int 数据类型。

2）强制类型转换时，是不改变原来变量的类型的，只是得到一个所需类型的中间变量，如上例中的变量 x 和变量 y，其本身的数据类型和值并没有发生变化。

【例 4.9】

```
main(){
   float f=1.39;
   printf("(int)f=%d,f=%f\n",(int)f,f);
}
```

从以上程序执行结果可以看出，float 型变量 f 虽然在 printf 函数中强制类型转为 int 型，但它只在运算中起作用，而且是临时的，变量 f 其本身的类型并不改变。因此，（int）f 的值为 1（截去了小数部分数字），而 f 的值仍为 1.39。

4.2.2 算术运算符和算术表达式

C 语言中运算符和表达式数量之多，在高级语言中是少见的。正是丰富的运算符和表达式使 C 语言功能十分完善。这也是 C 语言的主要特点之一。

C 语言的运算符不仅具有不同的优先级，而且还具有另一个特点，那就是它的结合性。在表达式中，各运算量参与运算的先后顺序不仅要遵守运算符优先级别的规定，还要受到运算符结合性的制约，以便确定是自左向右进行运算还是自右向左进行运算。这种结合性在其他高级语言的运算符中是没有的，因此也在一定程度上增加了 C 语言的复杂性。

1. C 语言运算符简介

C 语言的运算符分为以下几类：

1）算术运算符：用于各类数值运算，包括加（+）、减（-）、乘（*）、除（/）、求余（或称模运算，%）、自增（++）、自减（--）共 7 种。

2）关系运算符：用于比较运算，包括大于（>）、小于（<）、等于（==）、大于等于（>=）、小于等于（<=）和不等于（!=）6 种。

3）逻辑运算符：用于逻辑运算，包括与（&&）、或（||）、非（!）3 种。

4）位操作运算符：参与运算的量，按二进制位进行运算，包括位与（&）、位或（|）、位非（~）、位异或（^）、左移（<<）、右移（>>）6 种。

5）赋值运算符：用于赋值运算，分为简单赋值（=）、复合算术赋值（+=，-=，*=，/=，%=）和复合位运算赋值（&=，|=，^=，>>=，<<=）3 类共 11 种。

6）条件运算符：这是一个三目运算符，用于条件求值（?:）。

7）逗号运算符：用于把若干表达式组合成一个表达式（,）。

8）指针运算符：用于取内容（*）和取地址（&）2 种运算。

9）求字节数运算符：用于计算数据类型所占的字节数（sizeof）。

10）特殊运算符：有括号（）、下标［］、成员（→，.）等几种。

2. 算术运算符和算术表达式

（1）基本的算术运算符

1）加法运算符“+”：如 a+b。

2）减法运算符/负数值符号“-”：如 a-b 或负数 -5。

3）乘法运算符“*”：如 a*b。

4）除法运算符“/”：如 a/b。

5）求余运算符（模运算符）“%”：比如 11%3=2，即 11 除以 3 后，余数为 2。

【例 4.10】

```
main(){
   printf("%d\n",110%3);
}
```

本例输出 110 除以 3 所得的余数 2。

（2）算术表达式和运算符的优先级和结合性

用算术运算符和括号将运算对象连接起来的，符合 C 语言语法的式子，称为算术表达式。其运算对象包括常量、变量、函数等。表达式求值运算按运算符的优先级和结合性来进行。

以下是算术表达式的例子：

a+c;

a+b/d*c;

a*(b-c)+(d+e)/f;

a-b/c-9.5+‘d’;

1）运算符的优先级：即当一个运算对象两边都有运算符时，执行运算的先后顺序。如优先级高的，则先进行运算。C 语言中，运算符的运算优先级共分为 15 个等级。1 级最高，15 级最低。在表达式中，优先级较高的在优先级较低的之前进行运算。而在一个运算对象两侧的运算符优先级相同时，那就得按运算符的结合性规定进行处理。

2）运算符的结合性：即当一个运算对象两边的运算符出现相等优先级情况下的运算顺序。C 语言中所有运算符的结合性分为两种，左结合性（自左向右）和右结合性（自右向左）。算术运算符的结合性是自左至右，即先左后右。

例如：a-b*c 从这个表达式可以看到，乘法的优先级要高于加减法，因此，先进行 b*c 的运算，然后再将其积加上 a。

(a-b)*(c+d)+e 该表达式比上面的表达式稍微复杂，因为括号在算术运算符中的优先级是最高的，故先做括号内的运算，即完成（a-b）和（c+d）内的运算，再将两个计算结果进行相乘，最后再加上 e。算术运算符的结合性是自左向右的，也称“左结合性”。而自右至左的结合方向称为“右结合性”。最常见的右结合性运算符是赋值运算符。例如 a=b=c，则应先执行 b=c 再执行 a=(b=c）运算。在编写程序的过程中，希望大家注意区别，以避免理解错误。

（3）强制类型转换运算符

一般形式为：

(类型名)　　(表达式)

它的功能是把表达式的运算结果强制转换成类型名所表示的数据类型。

例如：

(float) a　　把 a 转换为实型数据

(int) (x + y)　　把 x + y 的结果转换为整型数据

(4) 自增、自减运算符

++　增量运算符　其功能是使变量的值自增 1。

--　减量运算符　其功能是使变量值自减 1。

这两个运算符是 C 语言中所特有的，使用非常方便，其作用就是对变量做加 1 或减 1 操作。要注意的是变量在符号前或后，其含义都是不同的。

i++　　i 参与运算后，i 的值再自增 1。

i--　　i 参与运算后，i 的值再自减 1。

粗略地看，++i 和 i++的作用都相当于 i=i+1，但不同之处在于++i 先执行 i=i+1，再使用 i 的值；而 i++则是先使用 i 的值，再执行 i=i+1。

例如：

若 i 的值原来为 6，则

k=++i　k 的值为 7，i 的值也为 7

k=i++　k 的值为 6，但 i 的值为 7

若 i 的值为 3，表达式 k=(++i)+(++i)+(++i) 的值应该为 18，因为++i 最先执行，先对表达式进行扫描，进行 3 次自加 (++i)，则此时 i=6，之后，表达式应该体现为 k=6+6+6=18，所以才得出 18 这个数字。若表达式改为 k=(i++)+(i++)+(i++)，其最终的 k 值应该是 9，而 i 最终为 6，因此在这个表达式中，先对 i 进行 3 次相加，再执行 3 次 i 的自加。

【例 4.11】

```
main(){
int i=10;
printf("%d\n",++i);
printf("%d\n",--i);
printf("%d\n",i++);
printf("%d\n",i--);
printf("%d\n",-i++);
printf("%d\n",-i--);
}
```

i 的初值为 10，第 2 行 i 加 1 后输出后为 11；第 3 行减 1 后输出为 10；第 4 行输出 i 为 10，之后再本身加 1 (此时 i 为 11)；第 5 行输出 i 为 11，之后再本身减 1 (此时 i 为 10)；第 6 行输出 -10 之后再加 1 (此时 i 为 11)，第 7 行输出 -11 之后再减 1 (此时 i 为 10)。

3. 赋值运算符和赋值表达式

(1) 赋值运算符

赋值运算符“=”的作用就是将一个数据赋给一个变量。

其一般形式为：

变量=表达式

例如：

c=a+b

w = sin(x) + cos(y)

赋值表达式的作用是将表达式的计算结果赋给“ = ”左边的变量。由于赋值运算符具有右结合性。因此

如果有 x = y = z = 28

可理解为

x = (y = (z = 28))

（2）复合的赋值运算符

经常会看到一些源程序代码中，出现以下的运算符，这样做可以简化程序，提高 C 程序代码的编译效率。

+ =， - =， * =， / = ,% =， < < =， > > =，& =，^ =，| =

其效果例子如下：

a + = b　相当于 a = a + b

a - = b　相当于 a = a - b

a * = b　相当于 a = a * b

a/ = b　相当于 a = a/b

a% = b　相当于 a = a% b

a < < = b　相当于 a = a < < b

a > > = b　相当于 a = a > > b

复合赋值符这种写法；初学者可能会不太习惯，但它非常有利于编译处理，能提高编译效率并产生高质量的目标代码。

4. 逗号运算符和逗号表达式

在 C 语言中，提供了一种特殊的运算符，它就是逗号运算符“,”。用逗号运算符将两个表达式连接起来组成一个表达式，称为逗号表达式。

一般形式为：

表达式 1，表达式 2，表达式 3，…，表达式 n

逗号表达式的求解过程是：从左到右计算出各个表达式的值，而整个逗号运算表达式的值等于最右边那一个表达式的值，就是“表达式 n”的值。在大部分情况下，使用逗号表达式的目的只是为了分别得到各个表达式的值，而并不一定要得到使用整个逗号表达式的值。另外还要注意，并不是程序中任何位置上出现的逗号都可以认为是逗号运算符。如函数中的参数，同数据类型变量的定义中的逗号只是用来起到间隔作用，并不是逗号运算符。

【例 4.12】

```
main(){
int a = 1,b = 2,c = 3,x,y;
y = (x = a + b),(b + c);
printf("y = %d,x = %d",y,x);
}
```

例 4.12 中，y 等于整个逗号表达式的值，也就是（b + c）的值，x 是第一个表达式的值。对于逗号表达式的使用过程中还请注意以下两点：

1）逗号表达式可以进行嵌套使用。

例如：

表达式 1，（表达式 2，表达式 3）

可以把（表达式 2，表达式 3）看成是一个逗号表达式。

2）程序中使用逗号表达式，通常是要分别求逗号表达式内各表达式的值，但并不一定求得整个逗号表达式的值。

4.2.3　关系运算符和表达式

在程序中，经常会需要比较两个变量的大小关系，以便对程序的不同功能做出选择，把比较两个数据量的运算符称为关系运算符。

1. 关系运算符及其优先次序

在 C 语言中有以下关系运算符：

1）<　小于

2）< =　小于或等于

3）>　大于

4）> =　大于或等于

5）= =　等于

6）！ =　不等于

对于关系运算符，我们并不陌生，C 语言中有 6 种关系运算符，这些运算符的意义看上去也非常直观。即使从来没用 C 语言写过程序，也应该对前面 4 个关系运算符非常熟悉了。而“= =”在 VB 或 PASCAL 等语言中是用“=”，“！ =”则是用“not”。关系运算符都是双目运算符，其结合性均为左结合。关系运算符的优先级低于算术运算符，高于赋值运算符。在 6 个关系运算符中，<、< =、>、> =的优先级相同，高于= =和！ =，= =和！ =的优先级相同。

2. 关系表达式

当两个表达式用关系运算符连接起来时，将其称之为关系表达式。关系表达式通常是用来判别某个条件是否满足。要注意的是用关系运算符的运算结果只有“0”和“1”两种，也就是逻辑的真与假，当结果为“1”时，则表示指定的条件满足，不满足时的结果为“0”。

关系表达式的一般形式为：

表达式 关系运算符　表达式

例如：

a + b > c

a > 9

都是合法的关系表达式，由于表达式也可以是关系表达式，因此表达式也允许出现嵌套的情况。例如：

a < (b < c)

x！ = (y = = z)

关系表达式为“真”时，用“1”表示；表达式为“假”时，用“0”表示。

如：

2 > 1 的值为“真”，即为“1”。

(a = 3) > (b = 5) 由于 3 > 5 不成立，故其值为假，即为 0。

【例4.13】

```
main () {
    char c ='k';
    int i =2, j =4, k =6;
    float x =2e +6, y =0.65;
    printf ("%d,%d\n", 'a' +5 < c, -i -2 * j > =k +1);
    printf ("%d,%d\n", 1 < j < 5, x -5.25 < =x +y);
    printf ("%d,%d\n", i +j +k = = -2 * j, k = =j = =i +5);
}
```

在本例中打印出了各种关系运算符的值。以上部分printf语句中，字符变量是以它相应的ASCII码值参与运算的，对于含多个关系运算符的表达式，如语句中出现“k = =j = =i +5”的情况，则C编译器会根据运算符的左结合性，先执行k = =j，该式不成立，其值为“0”，再执行0 = =i +5，因为i =2，所以等式也不成立，故表达式值最终的值为“0”。

4.2.4　逻辑运算符和表达式

1. 逻辑运算符及其优先次序

关系运算符所能反映的是两个表达式之间的大小关系，而逻辑运算符则是用于求条件式的逻辑值，用逻辑运算符将关系表达式或逻辑量连接起来的就是逻辑表达式了。也许你会对为什么“逻辑运算符将关系表达式连接起来就是逻辑表达式了”这一个描述有疑惑的地方。其实在前面部分已经说过，用关系运算符的运算结果只有“0”和“1”两种，也就是逻辑的真和假，换句话说也就是逻辑量，而逻辑运算符就用于对逻辑量运算的表达。

C语言中提供了3种逻辑运算符：

1）&&　“与”运算

2）| |　“或”运算

3）!　“非”运算

与运算符“&&”和或运算符“| |”均为双目运算符，具有左结合特性。非运算符“!”为单目运算符，具有右结合特性。

逻辑运算符和其他运算符优先级的关系如图4-4所示。

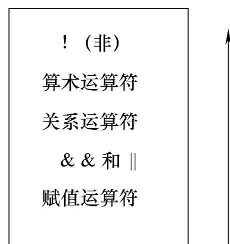


图4-4　运算符优先级

“&&”和“| |”低于关系运算符，“!”高于算术运算符。

同样逻辑运算符也有优先级别，从高到低排列，依次如下：!(逻辑非)→&&(逻辑与)→| |(逻辑或)，其中，逻辑非的优先级最高，逻辑或的优先级最低。

按照运算符的优先顺序可以得出：

a > b && x < y　等价于　(a > b)&&(x < y)

! b = =c||d < e　等价于　((! b) = =c)||(d < e)

a + b > c&&x + y < z　等价于　((a + b) > c)&&((x + y) < z)

再来看一个例子，如有　! True　| |　False && True

按逻辑运算的优先级别来分析则得到（True代表真，False代表假）

! True　| |　False　&&　True

False　| |　False　&&　True　//! Ture先运算得False

False　| |　False　　　　　//False　&&　True 运算得 False

False　　　　　　　　//最终 False　| |　False 得 False

2. 逻辑运算的值

逻辑运算的结果值分为“真”和“假”两种，用“1”和“0”来表示。求值规则如下：

1）与运算 &&：参与运算的两个量都为真时，结果才为真，否则为假。

例如：

4 >0 && 9 >2

因为 4 >0 为真，9 >2 也为真，两边同时满足真，所以它们相“与”的结果也为真。

2）或运算 | |：参与运算的两个量只要有一个为真时，结果就为真。两个量都为假时，结果为假。

例如：

4 >0| |5 >8

虽然 5 >8 为假，但因为 5 >0 为真，所以其最终结果也就为真。

3）非运算!：参与运算的量为真时，结果为假；参与运算的量为假时，结果为真。

例如：

!（5 >0）

其的结果应该为假。

虽然 C 语言在进行逻辑运算时，以“1”代表“真”，“0”代表“假”，但是否意味着“真”就是“1”，“假”就是“0”呢？其实在 C 语言中并不是这样的，C 语言规定，“0”代表“假”，而以非“0”值代表为“真”，这点请大家注意区别。

例如：

由于 1 和 9 均为非“0”因此 1&&9 的值为“真”，即为“1”。

又如：

9| |0 的值为“真”，即为“1”。

3. 逻辑表达式

逻辑表达式的一般形式为：

表达式 1　逻辑运算符　表达式 2

与关系表达式类似，逻辑表达式同样也可以出现嵌套的情况。

例如：

（x&&y）&&z

根据逻辑运算符的左结合性，上面式子等价于：

a&&b&&c

逻辑表达式的值就是式子中所有逻辑运算的最后结果值，用“1“和“0”分别代表“真”和“假”。

逻辑“与”，就是当条件式 1 与条件式 2 都为真时，结果为真（非 0 值），否则为假（0 值）。也就是说编译器会先对条件式 1 进行判断，如果为真（非 0 值），则继续对条件式 2 进行判断，当结果为真时，逻辑运算的结果为真（值为 1），如果结果为假时，逻辑运算的结果为假（0 值）。如果条件式 1 的逻辑值为假时，那就不用再判断条件式 2 了，而直接给出运算结果为假，即值为“0”。

逻辑“或”，是指只要两个运算条件中有一个为“真”时，运算结果就为“真”，只有当条件式都为假时，逻辑运算结果才为假。

逻辑“非”，则是把逻辑运算结果值取反，也就是说如果两个条件式的运算值为真，进行逻辑非运算后则结果变为假，条件式运算值为假时，那么最后逻辑结果为真。

【例 4.14】

```
main( ){
    char c ='k';
    int i =1,j =3,k =5;
    float x =3e +6,y =0.25;
    printf( "%d,%d\n",! x*! y,!!! x);
    printf( "%d,%d\n",x||i&&j-3,i < j&&x < y);
    printf( "%d,%d\n",i = =5&&c&&(j =8),x + y||i + j + k);
}
```

例 4.14 中！x 和！y 的值分别为 0，所以！x＊！y 也为 0，故其输出值为 0。由于 x 为非 0，故!!! x 对非 0 做了 3 次的逻辑非操作，最终的逻辑值为 0。对于式子 x||i && j-3，先计算 j-3 的值为 0，再求 i && j-3 的逻辑值为 0，但 x 为非 0，因此 x||i&&j-3 的逻辑值为 1。对于式子 i < j&&x < y，由于 i < j 的值为 1，而 x < y 为 0，故表达式的值为“1”和“0”相与，最终等于 0，对于式子 i = =5&&c&&（j =8），由于 i = =5 为假，即值为 0，该表达式由两个逻辑与运算组成，而当第一个表达式的值为“0”时，就不会再去判断后面的表达式的值了，所以整个表达式的值为 0。对于式子 x + y||i + j + k 由于 x + y 的值为非 0，故整个或表达式的值为 1。

第 5 章　分支与循环控制

5.1　if 语句

5.1.1　程序的 3 种基本结构

C 语言作为一种结构化的语言，是以模块为基本单位的，不允许出现交叉程序流程的存在。使用结构化程序的设计方法，使得程序结构清晰、易于维护和阅读。结构化程序由若干模块组成，每个模块中包含着若干个基本结构，而每个基本结构中可以有若干条语句。

C 语言具有 3 种基本结构：顺序结构、选择结构、循环结构。

（1）顺序结构及其基本流程

顺序结构是最简单、最基本的程序结构。程序由低地址向高地址顺序执行指令代码。如图 5-1 所示，程序先执行 A 操作，再执行 B 操作，两者是顺序执行的关系。

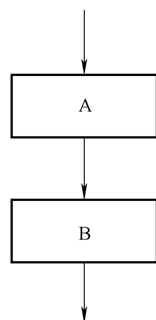


图 5-1　顺序结构流程图

（2）选择结构及其基本流程

选择结构使计算机拥有了判断的能力，或者说是决策的能力。如图 5-2 所示，如果条件判断为真，即条件满足，就执行 A，否则就执行 B。

（3）循环结构及其基本流程

循环结构有两种形式：当型循环结构和直到型循环结构。

当型循环结构，如图 5-3 所示。当条件成立时，反复执行 A，直到条件不满足为止。

直到型循环结构，如图 5-4 所示。先执行 A 操作，再进行判断，若为假，再执行 A，如此反复，直到条件为真为止。

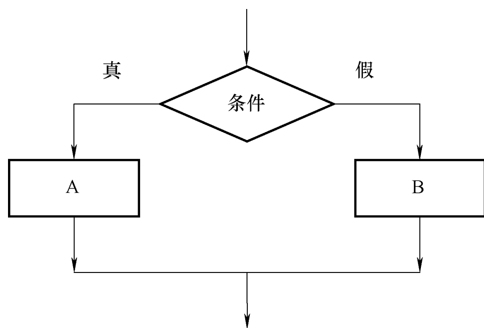


图 5-2　选择结构流程图

5.1.2　if 语句的 3 种形式

C 语言的条件语句与其他语言也是一样，“如果…就…”或是“如果…就…否则…”，也就是当条件符合时就执行语句。条件语句又被称为分支语句，它根据给定的条件进行判断，以决定执行某个分支程序段。也有人会称为判断语句，其关键字是由 if 构成，这在众多的高级语言中都是基本相同的。C 语言提供了 3 种形式的条件语句。

1．基本形式：if

if（表达式）语句

其中的表达式就是判断条件，如果是真就执行后面的语句，否则就不执行。其执行过程如图 5-5 所示。

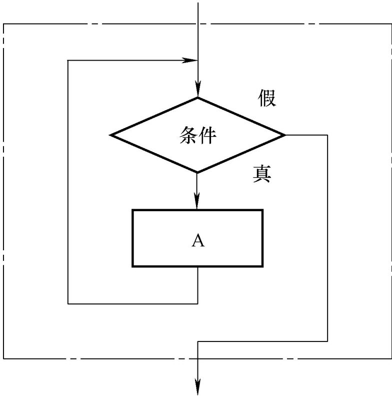


图 5-3　当型循环结构

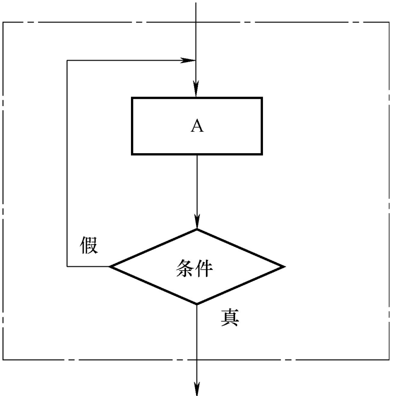


图 5-4　直到型循环结构

【例 5.1】

```
main()
{
    int a,b,max;
    printf("\n 请输入两个数：  ");
    scanf("%d%d",&a,&b);
    max = a;
    if (max < b) max = b;
    printf("最大的一个数为 = %d",max);
}
```

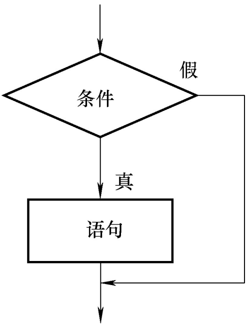


图 5-5　第 1 种 if 语句执行情况

这是一个简单的用 if 语句来判断两个数哪个比较大的程序，输入两个数 a、b。把 a 先赋予变量 max，再用 if 语句判别 max 和 b 的大小，如 max 小于 b，则把 b 赋予 max。所以 max 中放的总是大的那个数，最后将其值进行输出。

2. 第 2 种形式：if-else

```
if (表达式)
  语句 1;
else
  语句 2;
```

其意义为：如果表达式的值为真，则执行语句 1，否则执行语句 2 。其执行过程如图 5-6 所示。

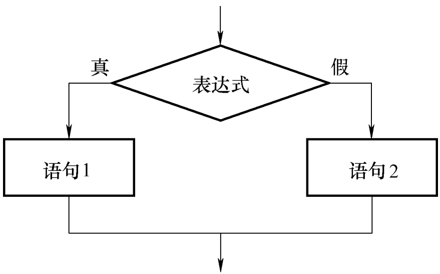


图 5-6　第 2 种 if 语句执行情况

【例 5.2】

```
main()
{
    int a,b,c;
    printf("请输入两个数");
    scanf("%d%d",&a,&b);
    if (a = =b)
```

```
    {
        c=1;
    else
        c=2;
    }
    printf("c=%d",c);
}
```

在该程序中，输入两个整数，用 if-else 语句判别，当 a 等于 b 时，c=1 否则，c=2，通过 printf 语句输出变量 c 的值。

3. 第 3 种形式：if-else-if

前面两种形式的 if 语句一般都用于两个分支的情况。当有多个分支选择时，可采用第 3 种形式，即 if-else-if 语句，其一般形式为：

```
if(表达式 1)
    语句 1;
else  if(表达式 2)
    语句 2;
else  if(表达式 3)
    语句 3;
    …
else  if(表达式 m)
    语句 m;
else
    语句 n;
```

其意义是：先判断表达式 1 的值，如果为真，则执行语句 1，如果表达式 1 的值为假，

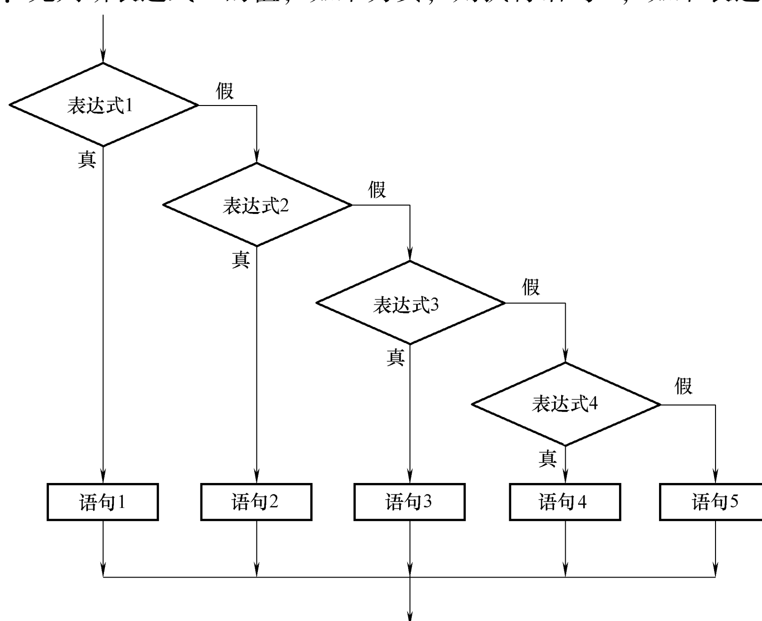


图 5-7　第 3 种 if 语句执行情况

则再判断表达式 2 的值，如果表达式 2 的值为真，则执行语句 2，否则继续判断表达式 3 的值，就这样依次判断表达式的值，当出现某个值为真时，则执行其后面对应的语句，语句执行完后跳到整个 if 语句之外继续执行程序代码。如果所有的表达式都为假，那么执行语句 n，即最后一个 else 后面的语句，然后再继续执行后面的程序代码。if-else-if 语句的执行过程如图 5-7 所示。

【例 5.3】

```
#include"stdio.h"
main()
{
    char c;
    printf("输入一个字符:");
    c=getchar();
    if(c<32)
      printf("这是一个控制字符\n");
    else if(c>='0'&&c<='9')
      printf("这是一个数字\n");
    else if(c>='A'&&c<='Z')
      printf("这是一个大写字母\n");
    else if(c>='a'&&c<='z')
    printf("这是一个小写字母\n");
    else
      printf("这是其他类型的字符\n");
}
```

该程序的功能是判断键盘输入字符的类型。其原理是根据输入字符的 ASCII 码值来判别字符类型。这是一个多分支选择问题的典型应用，用 if-else-if 语句来实现，判断输入的字符 ASCII 码所在的范围，分别给出不同的输出提示信息。例如输入 7，则程序会输出“这是一个数字”。

4. 使用 if 语句要注意的问题

1）在 3 种形式的 if 语句中，在 if 关键字之后均为表达式，它除了是常见的关系表达式或逻辑表达式外，也允许是其他类型的数据，如整型、实型、字符等。

例如：

if（a > 100）语句；

if（a）语句；

if（1）语句；

以上几种写法都是合法的，只要括号里的表达式为非 0，就会执行后面的语句，否则就不执行。

2）与 Basic 语言不同的是，在 if 语句中，条件判断语句必须用括号将表达式括起来，而且在语句后面必须加分号。

3）所有的 if 语句形式中，if 后面跟着的语句应为单条语句，如果想在满足条件分支时执行多条语句，那么必须把这若干条语句用“{}”括起来，称此为复合语句。C 语言中，

每一条语句末尾需要加上“;”，在复合语句中，需要注意的是，分号应加在“}”之内，而不能加在“}”外面。

例如：

```
if(x > y)
  {
    a + + ;
    printf("x > y");
  }
else
  {
    a - - ;
    printf("x < = y");
  }
```

在这段程序中，如果 x > y，变量 a 自加 1，打印输出“x > y”；如果 x < y 或 x = y，变量 a 自减 1，打印输出“x < = y”。

C 语句中有许多的括号，例如 {}，[]，()。这对于刚刚学习 C 语言的人来说，或许会很容易搞混。在 VB 等一些语言中同一个“()”会有不同的作用，但是在 C 语言中它们的分工是比较明确的。“{}”是用于将若干条语句组合在一起形成一种功能块，这种由若干条语句组合而成的语句就叫复合语句。复合语句之间用“{}”分隔，而它内部的各条语句还是需要以分号“;”结束。复合语句是允许嵌套的，也是就是在“{}”中的“{}”也是复合语句。复合语句在程序运行时，“{}”中的各行单语句是依次顺序执行的。C 语言中可以将复合语句视为一条单语句，也就是说，在语法上等同于一条单语句。对于一个函数而言，函数体就是一个复合语句，复合语句中不单可以用可执行语句组成，还可以用变量定义语句组成。要注意的是，在复合语句中所定义的变量，称为局部变量，所谓局部变量就是指它的有效范围只在复合语句中，而函数也算是复合语句，所以函数内定义的变量有效范围也只在函数内部。“[]”是用于数组的。“()”是用于写条件判断语句的。

5.1.3　if 语句的嵌套

在 if 语句里面，再写 if 语句，就是 if 语句嵌套。其一般表现形式如下：

```
if (表达式)
   if 语句;
```

或者为

```
if (表达式)
   if 语句;
else
   if 语句;
```

嵌套部分的 if 语句可能是简单的 if 类型，也有可能是 if-else 类型，甚至是复杂的很多层的 if-else-if 类型。这个时候就需要特别注意它们的层次关系，以及 if 和 else 的配对关系，要养成良好的程序编写习惯，层次分明，不仅易于阅读，而且可以避免出错。

下面来看一个例子：

```
if（表达式 1）
if（表达式 2）
    语句 1;
else
    语句 2;
```

大家现在想一想，上面这个程序，其中的 else 究竟是和哪个 if 是配对的呢？

或许是理解成这样：

```
if（表达式 1）
  if（表达式 2）
    语句 1;
  else
    语句 2;
```

else 是与第 2 个 if 配对的。

或者也可以这样理解：

```
if（表达式 1）
  if（表达式 2）
    语句 1;
  else
    语句 2;
```

else 是与第 1 个 if 配对的。

那么 else 究竟是和哪个 if 配对的呢？为了避免这种二义性，C 语言规定 else 语句总是与它前面最近的 if 相配对，因此对上述例子第 1 种情况理解是正确的。

【例 5.4】 比较两个数的大小关系。

```
main(){
    int x,y;
    printf("请输入两个数字:");
    scanf("%d%d",&x,&y);
    if(x!=y)
      if(x>y)
        printf("第一个数字比第二个数字大\n");
      else
        printf("第一个数字比第二个数字小\n");
    else
      printf("第一个数字等于第二个数字\n");
  }
```

该程序比较了两个数的大小关系，并通过了 if 语句的嵌套使用，判断了它们的 3 种关系：大于、小于、等于。

【例 5.5】 计算函数。

$y = 1 \quad x > 0$

$y = 0 \quad x = 0$

```
                y = -1    x<0
main( )
{
        float x,y;
        printf("请输入一个数:");
        scanf("%f" ,&x);
        if (x> =0)
           if (x>0)
                    y=1;
                else
                    y=0;
        else
           y= -1;
        printf("y=%f \n" ,y );
}
```

该程序使用了 if 嵌套语句来实现多个条件分支，从而完成函数的计算。可以看出，如果分支太多的话会使程序看起来比较混乱，所以一般有超过 3 个以上的分支，则更多的是使用另一个语句——switch 语句。

5.2　条件运算符和条件表达式

在前面学习了 if 语句，但是如果在条件语句中，只执行单个的赋值语句时，常可使用条件表达式来实现。

其一般格式为：

表达式 1?　表达式 2：表达式 3

其意义为：当逻辑表达式 1 的值为真（非 0 值）时，整个表达式的值为表达式 2 的值；当逻辑表达式的值为假（值为 0）时，整个表达式的值为表达式 3 的值。要注意的是条件表达式中逻辑表达式的类型可以与表达式 2 和表达式 3 的类型不一样。

其中的“?：”符号是三目运算符，它要求有 3 个操作对象，所以被称为三目运算符，也是 C 语言中唯一的三目运算符。

例如，有以下 if 语句：

```
if (x>y)
   max=x;
else
   max=y;
```

可用条件表达式可以简写为：

```
max= (x>y)? x: y;
```

其意义与上面这段 if 语句是一样的，很明显代码比上一段程序要少很多，编译的效率相对来说也就高一些，但有着和复合赋值表达式一样的缺点就是可读性相对较差。在实际应用时要根据自己习惯来使用。

使用条件表达式时，还应注意以下几点：

1）条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符。因此，

max = （x > y）？ x： y

可以去掉括号而等价为：

max = x > y？ x： y

2）条件运算符“？”和“：”是一个整体，即一个运算符，不能将其分开单独使用。

3）条件运算符的结合方向是自右向左的。

例如：

a < b？ a：c < d？ c：d

等价为

a < b？ a：（c < d？ c：d）

这种类似于 if 语句的嵌套使用，也就是条件表达式嵌套的情形，即其中的表达式 3 可以又是一个条件表达式。

【例 5.6】 输入一个字符。判别它是否是大写字母，如果是，将其转换为小写，否则不转换。然后输出最后得到的字符。

```
main( )
{
    char ch;
    scanf("%c",&ch);
    ch = (ch >= 'A'&& ch <= 'Z') ? (ch + 32):ch;
    printf("%c",ch);
}
```

5.3　switch 语句

前面已经提到了 switch 语句，如果程序有过多的分支时，一般采用 switch 语句，可以使程序结构清晰。其一般形式为：

```
switch (表达式)
  {
   case 常量表达式 1:     语句 1; break;
   case 常量表达式 2:     语句 2; break;
   …
   case 常量表达式 n:     语句 n; break;
   default          :     语句 n+1;
  }
```

其意义是：计算 switch 后面表达式的值，并将其作为条件与 case 后面的各个常量表达式的值相比，如果相等时则执行 case 后面的语句，再执行 break（间断语句）语句，跳出 switch 语句结构；如果 case 后面没有和条件相等的值时就执行 default 后的语句。如果当没有符合的条件时，不做任何处理，那可以不写 default 语句，default 语句只是程序不满足所有 case 语句条件情况下的一个默认情况执行语句。

【例 5.7】

```
main()
{
        int month;
        printf("请输入当前的月份:");
        scanf("%d",&month);
        switch (a)
        {
             case 1:printf("一月\n");
             case 2:printf("二月\n");
             case 3:printf("三月\n");
             case 4:printf("四月\n");
             case 5:printf("五月\n");
             case 6:printf("六月\n");
             case 7:printf("七月\n");
             case 8:printf("八月\n");
             case 9:printf("九月\n");
             case 10:printf("十月\n");
             case 11:printf("十一月\n");
             case 12:printf("十二月\n");
             default:printf("错误的月份数！\n");
       }
}
```

该程序使用了 switch 语句，可以看出这个程序拥有 13 个分支，如果使用 if 语句的话会显得非常混乱，但是使用了 switch 语句，就感觉非常清晰明了。但是当输入“3”之后，却执行了 case3 以及以后的所有语句，输出了“三月”及以后的所有单词，这当然并非我们的本意所在。为什么会出现这种情况呢？这恰恰反映了 switch 语句的一个特点。在 switch 语句中，“case 常量表达式”只相当于一个语句标号，表达式的值和某标号相等则转向该标号执行，但不能在执行完该标号的语句后自动跳出整个 switch 语句，所以出现了继续执行所有后面 case 语句的情况。这是与前面介绍的 if 语句完全不同的，应特别注意。为了避免上述情况，C 语言还提供了一种 break 语句，专门用于跳出 switch 语句，break 语句只有关键字 break，没有参数。在后面还将详细介绍。修改例题的程序，在每一条 case 语句之后增加 break 语句，使每一次执行之后均可跳出 switch 语句，从而避免输出不应有的结果。

【例 5.8】　输入月份，打印 1999 年某月有几天。

程序如下：

```
#include <stdio.h>
main( )
{
        int month;
        int day;
```

```
        printf("please input the month number :");
        scanf("%d" , &month);
        switch (month)
        {
          case 1:
          case 3:
          case 5:
          case 7:
          case 8:
          case 10:
          case 12: day =31;
               break;
          case 4:
          case 6:
          case 9:
          case 11:day =30;
               break;
          case 2: day =28;
               break;
          default : day = -1;
        }
        if day = -1
               printf("Invalid month input ! \n");
        else
        printf("1999. %d has %d days \n",month,day);
}
```

使用 switch 语句时应注意以下几点：

1）在 case 后的各常量表达式的值都应该是不一样的，否则会出现错误。

2）在 case 后，允许出现多条语句，可以不用 {} 括起来。

3）各 case 和 default 语句位置的先后顺序可以改变，而不会影响程序执行结果。

4）default 子句可以省略不写。

程序举例

【例 5.9】 输入 3 个整数，输出最大数和最小数。

```
main()
{
        int a,b,c,max,min;
        printf("input three numbers:");
        scanf("%d%d%d",&a,&b,&c);
        if(a > b)
           {max = a;min = b;}
```

```
        else
            {max = b;min = a;}
        if(max < c)
            max = c;
        else
            if(min > c)
        min = c;
        printf("max = %d\nmin = %d",max,min);
}
```

【例 5.10】 计算器程序。用户输入运算数和四则运算符，输出计算结果。

```
main()
{
        float a,b;
        char c;
        printf("input expression: a + ( - , * ,/)b \n");
        scanf("%f%c%f",&a,&c,&b);
        switch(c)
        {
            case'+': printf("%f\n",a + b);break;
            case'-': printf("%f\n",a - b);break;
            case'*': printf("%f\n",a * b);break;
            case'/': printf("%f\n",a/b);break;
            default: printf("input error\n");
        }
}
```

5.4 循环控制

5.4.1 概述

在现实生活中，有很多问题都需要用到循环控制。如一个频率为 12MHz 的 51 单片机应用电路中，要求实现 1ms 的延时，那它就要执行 1000 次空语句才可以达到延时的目的（当然可以使用定时器来做，后面将介绍定时器的使用），如果手工写 1000 条空语句那是非常麻烦的事情，难以想象，其次就是要占用很多的存储空间。我们知道这 1000 条空语句，都是一样的语句，即一条空语句重复执行 1000 次，因此就可以用循环语句去写，这样不但使程序结构简洁，查看代码显示直观，而且使其编译的效率大大提高。

C 语句中的循环语句有：

1）goto 语句和 if 语句构成循环；

2）while 语句；

3）do-while 语句；

4）for 语句。

5.4.2　goto 语句和 if 语句构成循环

goto 语句是无条件转换语句，在很多高级语言里都有，比如 BASIC 语言。它的一般形式为：

goto　语句标号；

其中的"语句标号"用标识符来表示，它的命名规则与变量名一样，即由字母、数字和下划线组成，第一个字符必须为字母或下划线。这个标识符加"："号出现在程序的某一行，执行到 goto 语句后，程序就会跳转到该标号处并执行其后的语句。标识符必须与 goto 语句在同一个函数里，否则无效。通常将 goto 语句与 if 语句一起用，即当满足什么条件的时候就跳转到某处语句。

但是，goto 语句会使程序层次不清，不易读懂，所以编写程序时尽量不要使用该语句。

【例 5.11】

```
void main( )
{
        int a;
        a =0;
        loop: a + +;
        if  (a = =100) goto   end;
        goto loop;
        end: ;
}
```

在该程序中，使用了 goto 语句，并且配合使用 if 语句，从而形成了循环的效果。使用了标识符"loop:"，表示循环的开始，"end:"标识程序的结束。不过这样的用法并不太常见，goto 语句用得最多的是用它来跳出多重循环，但是它只可以从内层循环跳到外层循环，不能从外层循环跳到内层循环。过多地使用 goto 语句会使程序结构不清晰，过多的跳转就使程序变得又像汇编语言的风格，而失去了 C 程序模块化的优点，所以不提倡使用。

5.4.3　while 语句

while 在英语里面的意思是"当……的时候"，因此 while 语句的作用也很好理解，就是当条件满足的时候，执行后面的语句，它的一般形式为：

while（表达式）语句

其中的表达式为循环条件，语句为循环体。判断表达式是否为真（非 0），则执行后面的语句，执行一次完成之后再次回到 while 后面的表达式，进行判断，如果为真，则重复执行语句，否则跳出循环。当条件一开始就为假时，那么 while 后面的循环体一次都不会被执行就退出整个循环。

while 语句执行过程可用图 5-8 表示。

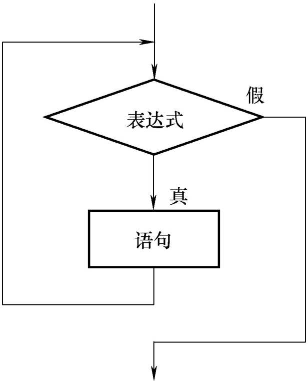


图 5-8　while 语句执行过程

【例 5.12】 用 while 语句求 $\sum_{n=1}^{100} n$ 。

用流程图和 N-S 结构流程图表示算法，如图 5-9 所示。

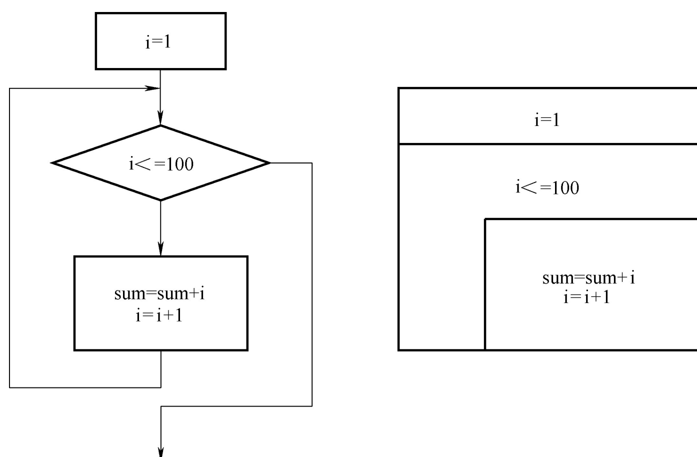


图 5-9　流程图及 N-S 图

```
main( )
{
        int i,sum =0;
        i =1;
        while(i < =100)
           {
                sum = sum + i;
                i + +;
           }
        printf("%d\n",sum);
}
```

执行结果：

5050

这个程序实现了 1 ~100 的累加，其中以下几点需要注意：

1）如果在第一次进入循环时，while 后圆括号内表达式的值为 0，循环一次也不执行。在本程序中，如果 i 的初值大于 100，将使表达式 i < =100 的值为 0，循环体也不执行。

2）在循环体中一定要有使循环趋向结束的操作，以上循环体内的语句 i + +使 i 不断增 1，当 i >100 时，循环结束。如果没有 i + +；这一语句，则 a 的值始终不变，循环将无限进行，进入死循环。

3）在循环体中，语句的先后位置必须符合逻辑，否则将会影响运算结果，例如，若将上例中的 while 循环体改写成：

```
while(i < =100)
{
     i + +;
     sum = sum + i;
}
```

运行后，将输出：5150，因为在程序运行的运行过程中，少加了第一项的值 1，而多加了最后一项的值 101。

5.4.4　do-while 语句

do-while 语句可以说是 while 语句的补充，while 是先判断条件是否成立再执行循环体，而 do-while 则是先执行循环体，再根据条件判断是否要退出循环。

do-while 语句的一般形式为：

```
    do
        语句
  while（表达式）;
```

在使用 do-while 语句有以下几点要注意：

1）do 是 C 语言的关键字，必须要与 while 联用。

2）do-while 循环是由 do 开始，到 while 结束。但是要注意的是，while（表达式）后面的"；"不能丢，它表示整个循环语句的结束。

3）while 后面的表达式是表示循环结束的条件，若为真则继续执行循环语句，否则结束循环。

其执行过程的流程图和 N-S 图，如图 5-10 所示。

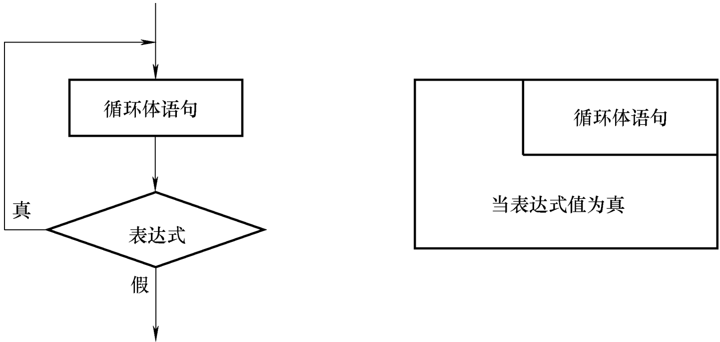


图 5-10　do-while 语句执行过程

【例 5.13】求 1～100 的累加和。

```
main（）
{
    int i, sum = 0;
    i = 1;
    do
      {
        sum = sum + i;
        i++;
      } while（i <= 100）;
    printf（"%d\n", sum）;
}
```

可以从这个程序看到，对于同一个问题可以用 while 语句处理，也可以用 do-while 处理。

它们之间可以互相转换。它们之间的主要区别是：while 循环的控制，出现在循环体之前，只有当 while 后表达式的值为非零时，才可能执行循环体；在 do-while 构成的循环中，总是先执行一次循环体，然后再求表达式的值，因此，无论表达式的值是零还是非零，循环体至少执行一次。和 while 循环一样，在 do-while 循环体中，一定要有能使 while 后表达式的值变为 0 的操作，否则，循环将会无限制地进行下去。

【例 5.14】 while 和 do-while 循环比较。

（1）main（）
```
{int sum =0,i;
scanf("%d",&i);
while(i< =20)
  {sum = sum + i;
   i++;
   }
  printf("sum = %d",sum);
  }
```

（2）main（）
```
{int sum =0,i;
scanf("%d",&i);
do
    {sum = sum + i;
     i++;
     }
  while(i< =20);
  printf("sum = %d",sum);
  }
```

5.4.5　for 语句

C 语言中的 for 语句使用非常灵活，它可以完全替代其他循环语句，不仅可以用于循环次数已定的情况下，而且在次数不定的情况下也可以使用。

它的一般形式为：

for（初值设定表达式；循环条件表达式；条件更新表达式）语句

它的执行过程如下：

1）先执行初值设定表达式。

2）然后求循环条件表达式的值，若它的值为真（非 0），则执行 for 语句中指定的内嵌语句，然后执行下面第 3 步；若值为假（0），则结束循环，转到第 5 步。

3）条件更新表达式。这个表达式是用来改变循

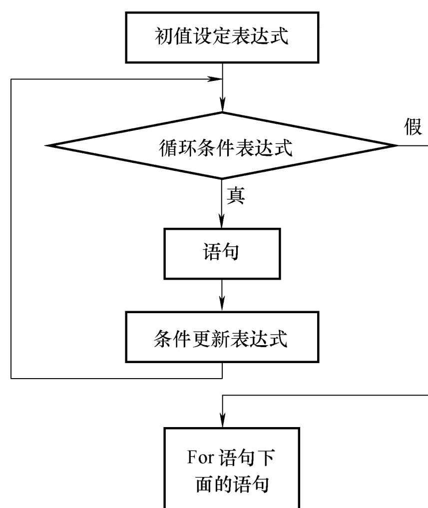


图 5-11　for 语句执行过程

环条件的，也是循环次数的控制。

4）转回上面第2步继续执行。

5）循环结束，执行 for 语句下面的语句。

该过程也可以用图5-11来表示。

例如：

```
for (i=1; i<=100; i++) sum=sum+i;
```

这段 for 语句实现了 1～100 的累加。先给变量 i 赋初值 1，判断 i 是否小于或等于 100，如果为真，则执行语句 “sum = sum + i”，然后 i 自增 1。然后再重新判断，直到条件为假，即 i > 100 时，循环结束。

相当于 while 语句：

```
i=1;
while(i<=100)
    {
        sum=sum+i;
        i++;
    }
```

对于使用 for 语句，要注意以下几点：

1）省略了“循环条件表达式”，如程序中不做相应的处理，则将变成为死循环。

例如：for (j=1;; j++) sum=sum+j;

2）如果不写“条件更新表达式”，则不对循环变量进行操作，这时可在语句体中加入修改循环变量的语句。

例如：

```
for(j=1;j<=100;)
  {
      sum=sum+j;
      j++;
  }
```

3）不写“初值设定表达式”和“条件更新表达式”。

例如：

```
    for(;j<=100;)
      {sum=sum+j;
       j++;}
```

相当于：

```
  while(j<=100)
      {sum=sum+j;
       j++;}
```

4）for 语句中的3个表达式可以全部省略不写。

例如：

```
    for (;;) 语句
```

相当于：

while(1)语句

即条件永远为真，形成死循环。

5）“初值设定表达式”可以是设置循环变量初值的表达式，也可以是其他表达式。

6）“初值设定表达式”和“条件更新表达式”可以是一个简单的表达式，也可以是好几个表达式，它们之间用逗号分隔。

例如：

```
for(j=0,i=1;i<=50;i++)j=j+i;
```

或者

```
for(i=0,j=100;i<=50;i++,j--)k=i+j;
```

7）“循环条件表达式”一般使用关系表达式或逻辑表达式，但也可是数值或字符的表达式，只要其最终的值为非零，即逻辑“真”，那么就执行循环体。

5.4.6　循环的嵌套

所谓循环的嵌套，就是指循环体里面包含了另一个完整的循环。与其他嵌套一样，要求层次要清楚，不能出现交叉的情况。

【例 5.15】求 1～5000 以内的完全数。

完全数：一个数如果恰好等于它的因子（除开自身）和，则称为完全数。例如，6 的因子为 1、2、3，而且 6=1+2+3，因此 6 是完全数。

```
main()
  {int i,j,s;
    for(i=1;i<=5000;i++)
        {s=0;
        for(j=1;j<i;j++)
          if(i%j==0)
            s+=j;
        if(i==s)
          printf("%6d\n",s);
        }
}
```

这个程序使用了两层的 for 循环嵌套。其中外层循环变量为 i，控制数据的取值范围；内层循环变量为 j，内层循环的循环体只有一条语句，用于求对应每一个 i 所有的因子和（s）（当 i 能被 j 整除时，j 就是 i 的一个因子）。当 i=s 时就输出该数。

几种循环的比较

到目前为止，介绍了好几种循环语句，虽然格式不同，但它们有着共同的特点，都是用于循环结构的程序设计。在程序设计的过程中，都具有如下 3 条内容：

1）循环体的设计。

2）循环条件的设计。

3）循环入口的初始化工作。

这 3 个条件都是缺一不可的，并且是一环扣一环的。循环体中安排哪些语句，要从分析具体问题入手，前后呼应，合乎逻辑，并且能确保循环能够终止，而且结论正确。

while、do-while 语句的使用，它的循环条件的改变，要靠程序员在循环体中去有意安排某些语句。而 for 语句却不必。使用 for 语句时，若在循环体中想去改变循环控制变量，以期改变循环条件，无异于画蛇添足。

while、do-while 循环适用于未知循环的次数的场合，而 for 循环适用于已知循环次数的场合。使用哪一种循环又依具体的情况而定。

凡是能用 for 循环的场合，都能用 while、do-while 循环实现，反过来未必能实现。

5.4.7　break 和 continue 语句

1. break 语句

在之前的 switch 语句学习中，我们已经接触了 break 语句。当 break 语句用于 do-while、for、while 循环语句中时，可使程序终止循环，跳出循环体而执行后面的语句，通常 break 语句与 if 语句一起使用，当满足条件时便跳出循环。

【例 5.16】 计算半径 r = 1 ~ 10 时的圆面积，直到面积大于 100 为止。

```
#define   PI   3.14159
main()
{
     float   r,area;
     for(r=1; r<=10; r++ )
        {
            area=PI*r*r ;
            if (area>100)   break;
            printf("%f",area);
        }
}
```

在该程序中，当 area > 100 时，使用了 break 跳出了循环，从而达到了程序目的。

2. continue 语句

continue 语句也是用来中断的语句，但是它与 break 语句不同，break 是用来跳出整个循环，而 continue 语句是跳出本次循环，也就是说出现在 continue 下面的语句将不再执行，直接进入下一次循环。

break 与 continue 的区别，如图 5-12 所示。

```
1) while（表达式 1）
        {……
          if(表达式 2)break;
          ……
        }
2) while（表达式 1）
          {……
          if（表达式 2）continue;
          ……
        }
```

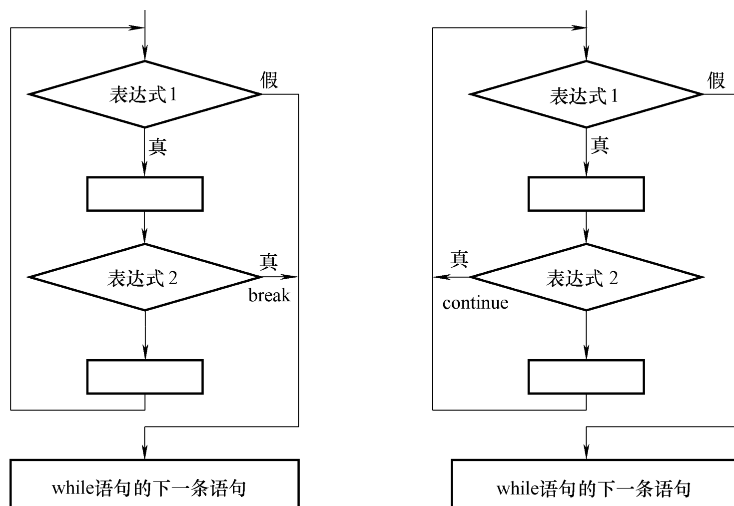


图 5-12　break、continue 语句执行过程比较

【例 5.17】　将 1 ~20 之间不能被 2 整除的数输出。

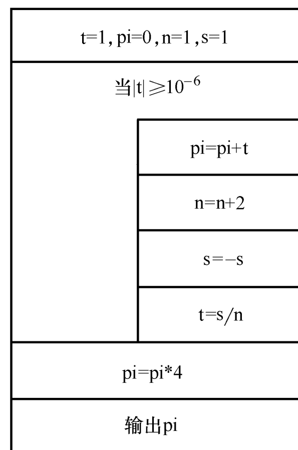
```
main()
{
        int i;
        for(i=1;i<=20;i++)
        {
            if (i%2==0) continue;
            printf("%d ",i);
        }
}
```

在该程序中，当变量 i 对 2 取模为 0 时，则为 i 能整除 2，此时执行 continue 语句，程序将跳到 for 语句继续执行，输出函数在此时将不会被执行。

程序举例

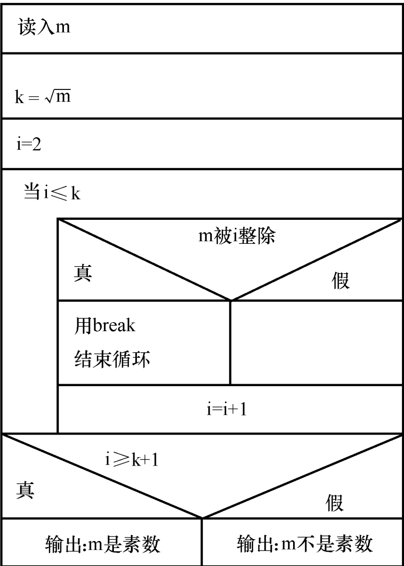
【例 5.18】用 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$ 公式求 π。

```
#include <math.h>
main()
{
      int s;
      float n,t,pi;
      t=1,pi=0,n=1,s=1;
      while(fabs(t)>1e-6)
      {pi=pi+t;
      n=n+2;
      s=-s;
      t=s/n;
```



```
    }
pi = pi * 4;
printf("pi = %10.6f\n",pi);
}
```

【例 5.19】判断 m 是否素数。




```
#include <math.h>
main()
{
int m,i,k;
scanf("%d",&m);
k = sqrt(m);
for(i = 2;i <= k;i ++)
   if(m%i == 0)break;
   if(i >= k + 1)
   printf("%d is a prime number\n",m);
   else
   printf("%d is not a prime number\n",m);
}
```

【例 5.20】求 100 ~200 间的全部素数。

```
#include <math.h>
main()
{
      int m,i,k,n = 0;
      for(m = 101;m <= 200;m = m + 2)
         {
            k = sqrt(m);
            for(i = 2;i <= k;i ++)
               if(m%i == 0)break;
               if(i >= k + 1)
                     {printf("%d",m);
                     n = n + 1;}
               if(n%n == 0)printf("\n");
         }
printf("\n");
}
```

第 6 章　编译预处理与位运算

6.1　概述

预处理是指在进行编译的第一遍扫描之前所做的工作，它由预处理程序负责完成，如包含命令# include、宏定义命令# define 等。在源程序中这些命令都放在函数之外，而且一般都放在源文件的前面，它们称为预处理部分。当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分进行处理，处理完毕后再进行编译工作。预处理命令不是 C 语言本身的组成部分，所以在使用时以“#”开头，以示和 C 语句的区别，在前面的章节中，我们已看到过多次以“#”开头的预处理命令。

C 语言提供了多种预处理功能，如宏定义、文件包含、条件编译等。合理地使用预处理功能编写的程序便于阅读、修改、移植和调试，也有利于模块化程序设计。本章介绍常用的几种预处理功能。

6.2　宏定义

“宏”是指用一个标识符来表示一个字符串。“宏名”就是指被定义为宏的标识符。在编译预处理时，对程序中所有的宏名，都用宏定义中的字符串去代替，称为宏代换。

宏定义由宏定义命令完成；宏代换由预处理程序完成。

在 C 语言中，宏定义有两种形式：不带参数的宏定义和带参数的宏定义。

6.2.1　不带参数的宏定义

定义的一般形式为：

#define　标识符　字符串

该宏定义的作用是出现标识符的地方均用字符串来替代。

如：#define PI 3.14

作用：用标识符（称为宏名）PI 代替字符串“3.14”。

宏展开：用定义的字符串去替换标识符，然后再对替换处理后的源程序进行编译，这一过程称为宏展开。在预编译时，将源程序中出现的宏名 PI 替换为字符串“3.14”，这一替换过程称为宏展开。

#define：宏定义命令

#undef：终止宏定义命令

【例 6.1】

```
#define PI 3.14
main()
{float l,s,r,v;
```

```
printf("input radius:");
scanf("%f",&r);/*输入圆的半径*/
l=2.0*PI*r;/* 圆周长*/
s=PI*r*r;/* 圆面积*/
v=4.0/3.0*PI*r*r*r;/* 球体积*/
printf("l=%10.4f\ns=%10.4f\nv=%10.4f\n",l,s,v);
}
```

注意事项：

1）宏定义必须以#define 开头，行末没有分号；

2）#define 命令一般出现在函数外部；

3）每一个#define 只能定义一个宏，且只占一行；

4）宏定义中的宏体只是一串字符，没有值和类型的含义，编译系统只对程序中出现的宏名用定义中的宏体作简单替换，而不作语法检查，且不分配内存空间；

5）宏体为空时，宏名被定义为字符常量 0。

宏定义的说明：

1）宏名一般用大写字母表示（变量名一般用小写字母）。

2）使用宏可以提高程序的可读性和可移植性。如上述程序中，多处需要使用 π 值，用宏名既便于修改又意义明确。

3）宏定义是用宏名代替字符串，宏扩展时仅作简单替换，不检查语法。语法检查在编译时进行。

4）宏定义不是 C 语句，后面不能有分号。如果加入分号，则连分号一起替换。

如：

```
#define PI 3.14;
area=P*r*r;
```

宏替换之后成为：

```
area=3.14;*r*r;
```

因此，在编译时会出现语法错误。

5）一般来说，通常把#define 命令放在一个文件的开头，使其在本文件全部有效（注意，#define 定义的宏仅在本文件有效，在其他文件中无效，这与全局变量不同）。

6）宏定义终止命令 #undef 结束先前定义的宏名。

```
#define G 9.8
main()
{
}
#undef G /* 取消 G 的意义*/
f1()
 ⋮
```

7）宏定义中可以引用已定义的宏名。

【例 6.2】

```
#define R 3.0
```

```
#define PI 3.14
#define L 2 * PI * R
#define S PI * R * R
main( )
{
printf( "L = %f\nS = %f\n" ,L,S) ;
}
```

8）对程序中用双引号括起来的字符串，即使与宏名相同，也不替换。如上例的 printf 语句中，双引号括起来 L 和 S 不被替换。

6.2.2　带参数的宏定义

定义的一般形式为：

#define　宏名（参数表）字符串

其含义是作相应的参数替换，带参数的宏在展开时，不是进行简单的字符串替换，而是进行参数替换，如图 6-1 所示。

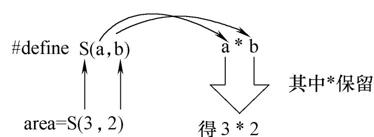


图 6-1　define 定义

【例 6.3】

```
#define PI 3.14
#define S(r)PI * r * r
main( )
{float a,area;
a =5;
area = S(a);
printf( "r = %f\narea = %f\n" ,a,area) ;
}
```

说明：

1）带参数的宏展开时，用实参字符串替换形参字符串，注意可能发生的错误。比较好的办法是宏定义的形参加括号，见表 6-1。

表 6-1　宏定义展开

宏　定　义	语　　句	展　开　后	
#define S(r)PI * r * r	area = S(a + b);	area = PI * a + b * a + b;	×
#define S(r)PI * (r) * (r)	area = S(a + b);	area = PI * (a + b) * (a + b)	√

2）宏定义时，宏名与参数表间不能有空格。

例如，#define S□(r)PI * r * r（□表示空格）

带参数的宏定义与函数的区别，见表 6-2。

表 6-2　带参数的宏定义与函数的区别

	函　　数	宏
信息传递	实参的值或地址传送给形参	用实参的字符串替换形参
处理时刻及内存分配	程序运行时处理，分配临时内存单元	宏展开在预编译时处理，不存在分配内存的问题

（续）

	函　　数	宏
参数类型	实参和形参类型一致。如不一致,编译器进行类型转换	字符串替换,不存在参数类型问题
返回值	可以有一个返回值	可以有多个返回值,见例 8.4
对源程序的影响	无影响	宏展开后使程序加长
时间占用	占用程序运行时间	占用编译时间

【例 6.4】 返回多个值的宏定义。

```
#define PI 3.14
#define CIRCLE(R,L,S,V)L=2*PI*R;S=PI*R*R;V=4/3*PI*R*R*R
main()
{float r,l,s,v;/*半径、圆周长、圆面积、球体积*/
scanf("%f",&r);
CIRCLE(r,l,s,v);
printf("r=%6.2f,l=%6.2f,s=%6.2f,v=%6.2f\n",r,l,s,v);
}
```

【例 6.5】 输出格式定义为宏。

```
#define PR printf
#define NL "\n"
#define A"%d"
#define L1 A NL
#define L2 A A NL
#define L3 A A A NL
#define L4 A A A A NL
#define S "%s"

main()
{int a,b,c,d;
char string[]="BOY";
a=1;b=2;c=3;d=4;
PR(L1,a);
PR(L2,a,b);
PR(L3,a,b,c);
PR(L4,a,b,c,d);
PR(S,string);
}
```

6.3　文件包含

所谓“文件包含”是指将一个源文件的全部内容包含到另一个源文件中去，成为另一个源文件中的一部分。文件包含预处理命令的一般格式为：

```
#include    < 文件名 >
```

或者

```
#include  "文件名"
```

以上两种格式的区别在于：前者，用< >括起文件名的文件包含命令，系统只在指定存放头文件的目录下（include 子目录下）查找该文件。后者，用双引号“”的包含命令，系统首先在源文件所在目录下查找该文件，如果找不到，再到指定存放头文件的目录下查找该文件。这里所说的“头文件”，因为#include 命令所指定的被包含文件常放在文件的开头，习惯上称被包含文件为头文件，并常以 h 作为其文件的扩展名。如“math. h”。用双引号“”的文件包含命令中的文件名还可改为文件路径，如“c：\ tc \ include \ math. h”。

“文件包含”命令是非常有用的，当一些共同的常量、数据等资料被用于多个程序时，就可以把这些共同的东西写在以 . h 作为扩展名的头文件中，如果有一个程序需要用到这些常量或数据时，就可用文件包含命令把它们包含进来，这样就省去了重复定义的麻烦。

例如：文件 file. c。

```
#include < stdio. h >
#define PI   3. 14
#define   AREA(r)   (PI * (r) * (r))
#define   PR   printf
#define   D   "%f"
```

为了省去重复定义的麻烦，以下的程序要用到以上内容，就可用文件包含命令把它们包含进来，形成一个新的源程序：

```
#include   "c：\tc\file. c"
main()
{float   r = 5,s;
s = AREA(5);
PR(D,s);}
```

注意：一个包含命令只能包含一个文件，若要包含 n 个文件，就需要 n 个包含命令。

6.4　条件编译

条件编译，顾名思义就是满足一定的条件下进行编译。在一般情况下，源程序中所有的行都要进行编译，但是有的时候只是希望对其中的一部分进行编译，这种条件编译对于提高 C 源程序的通用性是有好处的。例如，如果一个程序在不同的计算机系统上运行会有一定的差异（例如，有的机器以 16 位来存放一个整数，有的则是 32 位），这样往往需要对源程序作不断地修改，这样就降低了程序的通用性。

条件编译命令可以分为以下几种：

1. #ifdef

#ifdef 标识符程序段 1 #else 程序段 2 #endif	#ifdef 标识符 程序段 1 #endif

其中，“标识符”用#define 命令定义，#else 部分可以没有。

【例 6.6】　在调试程序时，常常需要输出一些信息，但在调试后又不需要再输出的，可以在源程序中插入以下的条件编译：

```
#define DEBUG
#ifdef DEBUG
printf("a = %d,b = %d,c = %d\n",a,b,c);
#endif
```

在程序调试完成后，如果不再需要显示 a、b、c 的值，则只需要去掉 DEBUG 标识符的定义。

当然，可以直接使用 printf（）语句显示调试信息，在程序调试完成后去掉 printf（）语句，也可以达到目的。但如果程序中有很多处需要调试观察，增、删语句既麻烦又容易出错。而使用条件编译则相当清晰、方便。

2. #ifndef

#ifndef 标识符 　程序段 1 　#else 　程序段 2 　#endif	#ifndef 标识符 　程序段 1 　#endif

与第 1 种形式不同：将“ifdef”改为“ifndef”。其作用是：若标识符未被定义，则编译程序段 1，否则编译程序段 2。这种形式正好与第 1 种形式相反。

3. #if

```
#if 表达式
程序段 1
#else
程序段 2
#endif
```

它的作用是：当表达式为真（非零）时就编译程序段 1，否则编译程序段 2。可以事先给定一定的条件，使程序在不同的条件下执行不同功能。

【例 6.7】　指定一串由字母组成的字符串，使用条件编译，使之字母全改为大写输出。

```
#define LETTER 1
main()
{
char str[20] = "I am a boy",c;
int i;
i = 0;
while((c = str[i])!= '\0')
{ i++;
#if LETTER
if(c >= 'a' && c <= 'z')
```

```
c = c -32;
#else
if(c > = 'A '&& c < = 'Z ')
c = c  + 32;
#endif
printf("%c",c);
}
}
```

运行结果为：I AM A BOY

若想改为小写字母，则将程序的第一行改为：#define LETTER 0 。

【例 6.8】

```
#define MAX 100
main()
{int i = 10;float x = 25.8;
#if MAX  >99
printf("%d\n",i);
#else
printf("%f\n",x);
#endif
printf("%d,%f\n",i,x);
}
```

运行结果 10

10,25.800000

若将 MAX 定义为 80,执行 printf("%f\n",x);

运行结果 25.800000

10,25.800000

6.5 位操作运算符

经过了前面的介绍，我们了解到各种运算是以字节为基本单位的，但是很多系统程序中通常要求在位（bit）一级进行运算或处理。C 语言能直接对硬件进行操作，因此就涉及位的概念，该功能使得 C 语言也能像汇编语言一样用来编写系统程序。

利用位操作运算符可对一个数按其二进制格式进行位操作。

例如，char A = 25，B = 77，将其化成二进制（或十六进制）表示，并作以下位操作运算。

1. 按位“与”运算符：&

C = A&B

A = 00011001(或 0x1B)

B = 01001101(或 0x4D)

C = 00001001(或 0x09)即 A = 9

& 可用作“掩码”运算，即屏蔽掉某些位。例如，掩码为 127，转换成二进制表示为 01111111，可屏蔽掉第 7 位。

2. 按位“或”运算符：|

C = A|B

A =00011001(或 0x1B)

B =01001101(或 0x4D)

C =01011101(或 0x5D)即 C =93

3. 按位“异或”运算符：^

C = A^B

A =00011001(0x1B)

B =01001101(0x4D)

C =01010100(0x54)即 C =84

很容易验证:C = A^B^B = A。

可以利用这个特性作一些数据的处理。例如，对一段文字进行加密，把它跟一个关键字进行异或处理，需要解密时，只需用同一个关键字再做一次异或处理，就可以使其恢复原样。再举一个例子，用下列异或处理程序：

```
swap1(a,b)
int a,b;
{
a = a^b;
b = a^b;
a = a^b;
}
```

代替下列程序：

```
swap(a,b)
int a,b;
{
int temp;
temp = a;
a = b;
b = temp;
}
```

同样实现 a 和 b 交换,异或处理的速度更快。

4. 按位取反运算符：~

F2 = ~F1，F2 为 F1 二进制按位取反。

例如：unsigned int F1 =0122457，F2；

二进制表示（八进制表示）

F1：1010010100101111（0122457）

F2：0101101011010000（0055320）

5. 左移运算符：<<

左移运算符“<<”是双目运算符。其功能把“<<”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补0。

例如：

设 a=3，a<<4

指把 a 的各二进位向左移动4 位。如 a=00000011（十进制3），左移4 位后为00110000（十进制48）。

6. 右移运算符：>>

右移运算符“>>”是双目运算符。其功能是把“>>”左边的运算数的各二进位全部右移若干位，由“>>”右边的数指定移动的位数。对于有符号数，在右移时，符号位将随同移动。当为正数时，最高位补0，而为负数时，符号位为1，最高位是补0 或是补1 取决于编译系统的规定。Turbo C 和很多系统规定为补1。

例如：

设　a=15，a>>2

表示把000001111 右移2 位，为00000011（十进制3）。

【例6.9】

```
main()
{
    unsigned a,b;
    printf("input a number:  ");
    scanf("%d",&a);
    b=a>>5;
    b=b&15;
    printf("a=%d\tb=%d\n",a,b);
}
```

再举一例：

【例6.10】

```
main()
{
    char a='a',b='b';
    int p,c,d;
    p=a;
    p=(p<<8)|b;
    d=p&0xff;
    c=(p&0xff00)>>8;
    printf("a=%d\nb=%d\nc=%d\nd=%d\n",a,b,c,d);
}
```

第 7 章　数组与函数

具有相同类型和名称的变量的集合，称之为数组。其中的变量称为数组的元素，每个数组元素都有一个编号，叫做下标，可以用一个统一的数组名和下标来唯一地确定数组中的元素。数组元素的个数称之为数组的长度。在 C 语言中，数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是各种基本数据类型或是构造类型。根据数组元素的类型不同，数组可以分为数值数组、字符数组、指针数组、结构数组等各种类别。

7.1　一维数组的定义和引用

7.1.1　一维数组的定义方式

所谓数组，可以举个例子来说明：这就好像学校操场上队列，每一个年级代表一个数据类型，每一个班级为一个数组，每一个学生就是数组中的一个元素。数组中的每个数据都可以用唯一的下标来确定其所在位置，下标可以是一维或多维的。比如在学校的方队中要找一个学生，这个学生是 I 年级 H 班 X 组 Y 号，那么可以把这个学生看作在 I 类型的 H 数组中（X，Y）下标位置中。数组和普通变量一样，要求先定义再使用，下面是定义一维或多维数组的方式：

一维数组的定义方式为：

类型声明符 数组名［常量表达式］；

其中，“类型声明符”是指数组中的各数据单元的类型，一个数组里的数据单元只能是同一数据类型。“数组名”是整个数组的标识，命名方法和变量命名方法是一样的。在编译时系统会根据数组大小和类型为变量分配空间，数组名可以说就是所分配空间首地址的标识。“常量表达式”是表示数组的长度和维数，它必须用“［ ］”括起，方括号里的数不能是变量只能是常量。

例如：

int m[10]；　　　　声明整型数组 m，有 10 个元素。

float a[15],b[15]；　　　　声明实型数组 a 和实型数组 b，各有 15 个元素。

char str[20]；　　　　声明字符数组 str，有 20 个元素。

注意，在 C 语言中数组的下标是从 0 开始的，而不像有些编程语言那样是从 1 开始的，如定义了 int a[10]，它的下标就是从 a[0] 到 a[9]，如引用单个元素就是数组名加下标，如 a[1] 就是引用 a 数组中的第 2 个元素，如果错用了 a[10] 就会出现错误。还有一点要注意的就是在程序中只有字符型的数组才可以一次引用整个数组，其他类型的则需要逐个引用数组中的元素，不能一次引用整个数组。

对于数组的使用应该要注意以下几点：

1）对于同一个数组，其所有元素的数据类型都是相同的，其类型都是根据数组被定义时的数据类型所定。

2）在取数组名时要注意不能与其他变量名同名。

例如：

```
  main( )
{
  float a;
  int a[10];
   ⋮
}
```

这样的定义是错误的。

3）“[ ]”中常量表达式表示数组元素的个数，如 a[10] 表示数组 a 有 10 个元素。10 个元素分别为 a[0]，a[1]，a[2]，a[3]，a[4] ，a[5] ，a[6] ，a[7]，a[8]，a[9]。

4）“[ ]”中的表达式可以是符号常数或常量表达式，但不可以是变量。

例如：

```
#define AA 10
main( )
{
float a[2 + 11],b[20 + AA];
 ⋮
}
```

这样的写法是合法的。

但是下面这样的写法是错误的。

```
main( )
{
  int b = 10;
  int a[b];
   ⋮
}
```

5）C 语言允许在同一个类型声明中，声明多个数组和多个变量。

例如：

```
int a,b,c,d,m[20],n[30];
```

7.1.2　一维数组元素的引用

一个数组被定义后，数组中的各个元素就共用一个数组名（ 即该数组变量名），就以它们的下标来区别各个元素。对数组的操作归根到底就是对数组元素的操作。

数组元素的表现形式为：

数组名［下标］

其中下标只能为整型常量或整型表达式。如为小数时，C 编译将自动取整。

例如：

```
a[10]
a[i + j]
```

a[j++]

这些都是合法的数组元素。

数组元素也被称为下标变量，必须先定义再使用下标变量。下标的值不允许超越所定义的下标下界和上界。

例如，一个数组有20个元素，将其各元素值逐个输出：

```
for(i=0;i<20;i++)
printf("%d",a[i]);
```

如改成下面的写法，则会出错，因为C语言中不能用一个语句输出整个数组，只能逐个输出。

```
printf("%d",a);
```

【例7.1】

```
main()
{
   int i,a[10];
   for(i=0;i<=9;i++)
       a[i]=i;
   for(i=9;i>=0;i--)
       printf("%d ",a[i]);
}
```

【例7.2】

```
main()
{
   int i,a[10];
   for(i=0;i<10;)
       a[i++]=i;
   for(i=9;i>=0;i--)
       printf("%d",a[i]);
}
```

【例7.3】

```
main()
{
   int i,a[10];
   for(i=0;i<10;)
       a[i++]=2*i+1;
   for(i=0;i<=9;i++)
       printf("%d ",a[i]);
   printf("\n%d %d\n",a[6.2],a[7.8]);
}
```

在这个例子中，用了一个循环语句给数组a各元素送入奇数数字，然后在第2个循环语句中，逐个输出奇数。在第1个for语句中，表达式3省略了，因为在下标变量中使用了表

达式 i + +，用来修改循环变量的值。程序最后的一个 printf 语句输出了 a［6］和 a［7］的值，因为当下标不为整数时编译器将自动取整。

7.1.3 一维数组的初始化

数组定义后，数组元素的值是随机的，可以用 3 种方法给数组元素赋值：

1）用赋值语句给数组元素赋值。

2）用输入函数从键盘或数据文件中读取数据并赋给数组元素。

3）初始化数组，即在定义数组的同时，为数组元素赋值。

前两种方法占用程序执行时间较多，是在程序的运行阶段实行的，每运行一次程序，相关的语句都必须执行一遍。最后一种方法对于静态存储的数组是在程序编译阶段完成赋初值的，只要程序编译成功，运行程序时不再执行相关的语句，因此减少了程序运行时间。另外，前两种方法对没有赋值的数组元素来说，数组元素的值是不确定的，第 3 种方法系统对没有赋初值的数组元素自动赋予一个确定值（整型或实型数组元素赋数值 0，字符型数组元素赋字符常量‘\0’。）

初始化赋值的一般形式为：

类型声明符 数组名[常量表达式]={初值,初值,…,初值}；

其中在 {} 中的各个数据为各数组元素的初值，各值之间用逗号分隔。

例如：

```
int a[5]={ 0,1,2,3,4 };
```

相当于 a[0]=0;a[1]=1…a[4]=4;

对数组元素的初始化可以用以下方法实现：

1）在定义数组时对数组元素赋初值。例如：

```
int a[5] = {0,1,2,3,4};
```

2）可以只给一部分元素赋值。例如：

```
int a[5] = {0,1,2};
```

3）对全部元素赋初值时，可以不指定数组长度。例如：

```
int a[5] = {0,1,2,3,4};
```

可以写成

```
int a[ ] = {0,1,2,3,4};
```

7.1.4 一维数组程序举例

在程序执行过程中，对数组作动态赋值。

【例 7.4】

```
main()
{
   int i,max,a[10];
   printf("input 10 numbers:\n");
   for(i=0;i<10;i++)
      scanf("%d",&a[i]);
   max=a[0];
```

```
    for(i=1;i<10;i++)
        if(a[i]>max)max=a[i];
    printf("maxnum=%d\n",max);
}
```

本例程序中第 1 个 for 语句中，使用了 scanf 函数逐个输入 10 个数到数组 a 中。然后把 a［0］赋值给 max。在第 2 个 for 语句中，从 a［1］到 a［9］逐个与 max 中的值比较，若比 max 的值大，则把该下标变量送入 max 中，因此 max 总是在已比较过的下标变量中为最大者。比较结束，输出 max 的值。

【例 7.5】

```
main()
{
    int i,j,p,q,s,a[10];
    printf("\n input 10 numbers:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    for(i=0;i<10;i++)
    {   p=i;q=a[i];
        for(j=i+1;j<10;j++)
        if(q<a[j]){ p=j;q=a[j];}
        if (i!=p)
          {s=a[i];
           a[i]=a[p];
           a[p]=s;}
        printf("%d",a[i]);
     }
}
```

本例程序中用了两个并列的 for 循环语句，在第 2 个 for 语句中又嵌套了一个循环语句。第 1 个 for 语句用于输入 10 个元素的初值。第 2 个 for 语句用于排序。本程序的排序采用逐个比较的方法进行。在 i 次循环时，把第 1 个元素的下标 i 赋予 p，而把该下标变量值 a［i］赋予 q。然后进入小循环，从 a［i+1］起到最后 1 个元素止逐个与 a［i］作比较，有比 a［i］大者则将其下标送 p，元素值送 q。一次循环结束后，p 即为最大元素的下标，q 则为该元素值。若此时 i≠p，说明 p、q 值均已不是进入小循环之前所赋之值，则交换 a［i］和 a［p］之值。此时 a［i］为已排序完毕的元素。输出该值之后转入下一次循环。对 i+1 以后各个元素排序。

7.2　二维数组的定义和引用

7.2.1　二维数组的定义

之前介绍了一维数组，它只有一个下标，其数组元素也称为单下标变量。

但是在现实生活中，仅仅使用一维数组，很多事物都无法恰当地被表示。举个例子：假如一个班级 40 个学员，把他们编成 1 ~ 40 号。但现在有两个班级要管理怎么办？每个班级都各有各的编号，比如 1 班学生编是 1 ~ 40；2 班的学生也是 1 ~ 40。现在两个班的学生编号要混在一起输入计算机系统，从 1 号编到 80 号，显然不是很合适，也很难进行有效的管理。在实际问题中有很多量是二维的或多维的，为了解决这个问题，C 语言允许构造多维数组。多维数组元素有多个下标，以标识它在数组中的位置。在这里，主要介绍一下二维数组，类似的还有三维、四维等，原理都是一样的，留给读者自己去思考。

二维数组定义的一般形式是：

类型声明符 数组名［常量表达式 1］［常量表达式 2］

其中常量表达式 1 表示第 1 维大小，常量表达式 2 表示第 2 维大小。

例如：

int a[3][4]；

说明了一个 3 行 4 列的数组，数组名为 a，其下标变量的类型为整型。该数组的数组元素共有 3 ×4 个，即

a[0][0],a[0][1],a[0][2],a[0][3]
a[1][0],a[1][1],a[1][2],a[1][3]
a[2][0],a[2][1],a[2][2],a[2][3]

由于数组 a 为整型，整型数据占 2 个字节的内存空间，所以数组中每个元素均占有两个字节。

7.2.2　二维数组元素的引用

二维数组的表示形式为：

数组名[下标 1][下标 2]

其中下标应为整型常量或整型表达式。

例如：

a[7][8]

表示 a 数组有 7 行 8 列元素。

这里的下标变量和一维数组定义时的元素个数有些相似，但这两者具有不同的含义。数组定义时方括号中给出的是某一维的长度，即长度的最大值；而数组元素中的下标变量是该元素在数组中的位置标识。注意：前者只能是常量，后者可以是常量、变量或表达式。

【例 7.6】　一个年级有 4 个班，有 3 门课的考试成绩的平均成绩，见表 7-1。求全年级分科的平均成绩和总平均成绩。

表 7-1　成绩分布表

	一班	二班	三班	四班
语文	80	70	75	85
数学	75	65	80	87
英语	92	71	70	90

可设一个二维数组 a［4］［3］存放 4 个班 3 门课的成绩。再设一个一维数组 v［3］存放所求得各分科平均成绩，设变量 average 为全年级总平均成绩。编程如下：

```
main()
{
   int i,j,s=0,average,v[3],a[4][3];
   printf("input score\n");
   for(i=0;i<3;i++)
   {
      for(j=0;j<4;j++)
      { scanf("%d",&a[j][i]);
         s=s+a[j][i];}
      v[i]=s/4;
      s=0;
   }
   average =(v[0]+v[1]+v[2])/3;
   printf("语文:%d\n 数学:%d\n 英语:%d\n",v[0],v[1],v[2]);
   printf("平均分:%d\n",average );
}
```

程序中首先用了一个双重循环。在内循环中依次读入某一门课程的各个班级的平均成绩，如第 1 次循环时，读入每个班的语文平均成绩，同时把所有班的语文成绩累加起来，退出内循环后再把该累加成绩除以 4 送入 v [i] 中，这就是该门课程的平均成绩。因为总共统计 3 门课程，所以外循环共循环 3 次，分别求出 3 门课各自的平均成绩并存放在 v 数组中。退出外循环之后，把 v [0]、v [1]、v [2] 相加除以 3 即得到各科总平均成绩。最后按题意输出全年级总平均成绩。

7.2.3　二维数组的初始化

二维数组初始化可以在数组定义时，给各下标变量赋以初值。

例如，对数组 a [4] [3]：

1）按分段进行赋值为：

```
int a[4][3]={ {80,75,92},{70,65,71},{75,80,70},{85,87,90}};
```

2）按连续进行赋值为：

```
int a[4][3]={ 80,75,92,70,65,71,75,80,70,85,87,90};
```

其最终效果都是一样的。

对于二维数组初始化赋值还要注意几点：

1）可以只对部分元素赋初值，未赋初值的元素自动取 0 值。

例如：

```
int a[2][2]={{5},{7}};
```

是对第 1 行第 1 列和第 2 行第 1 列元素赋值，其他元素为 0。语句执行后各元素的值为：

```
5 0 0
7 0 0
int a [3][3]={{1,2},{0,5,9},{3}};
```

赋值后的元素值为：

1 2 0

0 5 9

3 0 0

2）如要对所有数组元素赋初值，那么定义时，第 1 维的长度可以不写。

例如：

```
int a[3][3] = {9,8,7,6,5,4,3,2,1};
```

可以改为：

```
int a[][3] = {9,8,7,6,5,4,3,2,1};
```

7.3　字符数组

字符数组是用来存放字符的数组。

7.3.1　字符数组的定义

在前面所使用的例子中，用的数据是整型的，所以数组是一个整型数组。同样也可以将数组定义为其他类型，浮点型 float、布尔型 bool 或者字符型 char 均可以有相关的数组，如：

```
int age[] = {30,32,59,78};
float money[] = {500.50,7000,6500.75};
bool   married[] = {false,true};
char name[] = {'M','i','k','e'};
```

最后一行定义了一个字符数组，用来存储一个英文人名：Mike。

字符数组也可以是二维或多维数组。

例如：

```
char c[4][3];
```

即为二维字符数组。

7.3.2　字符数组的初始化

与整型数组一样，字符数组也可以在定义时作初始化赋值。

例如：

```
char a[8] = {'p','r','o','g','r','a','m'};
```

赋值后各元素的值为：

c[0]的值为'p'

c[1]的值为'r'

c[2]的值为'o'

c[3]的值为'g'

c[4]的值为'r'

c[5]的值为'a'

c[6]的值为'm'

当对全体元素赋初值时也可以省去长度说明。c［7］为‘\0’，作为字符串结束的标志。

例如：

```
char c[ ] = {'p','r','o','g','r','a','m'};
```

这时c数组的长度自动定为7。

7.3.3　字符数组的引用

【例7.7】

```
main()
{
   int i,j;
   char a[][5] = {{'C','h','i','n','a',},{'J','a','p','a','n'}};
   for(i=0;i<=1;i++)
     {
       for(j=0;j<=4;j++)
            printf("%c",a[i][j]);
       printf("\n");
     }
}
```

在该程序中二维字符数组a在初始化时将全部元素赋以初值，因此一维下标的长度可以不写，然后通过循环语句将各字符一一输出。

7.3.4　字符串和字符串结束标志

在C语言中通常用一个字符数组来存放一个字符串。前面也曾提到过字符串总是以'\0'作为结束标记。由此，在把一个字符串存入数组时，也把结束符'\0'存入了数组，以此作为该字符串结束的标志。

除了定义字符数组时初始化各元素值，还可以用字符串的形式来进行赋值。

例如：

```
char c[ ] = {'p','r','o','g','r','a','m'};
```

可写为：

char c[ ] = {"program"};　或 char c[ ] = "program";

用字符串方式赋值比用字符逐个赋值要多占一个字节，用于存放字符串结束标志'\0'。上面的数组c在内存中的实际存放情况为：

p	r	o	g	r	a	m	\0

'\0'是程序在编译时，由编译器自动加上的。由于有了'\0'标志，所以在用字符串赋初值时一般无须指定数组的长度，而由系统自行处理。

7.4　函数概述

对于一个较大的、较复杂的问题，人们通常将其分为若干个模块，每一个模块用来实现一个功能。C语言中的函数就是用来实现模块化程序设计的工具，相当于其他高级语言中的

子程序和过程。C 语言系统本身提供了极为丰富的库函数，同时还允许用户自己建立自定义函数。用户可以把自己的程序写成一个一个相对独立的函数，在需要用到的时候来调用它。换句话说，C 语言其实就是函数的集合，由于采用了函数结构的写法，使得 C 语言的程序结构清晰，同时有利于程序的阅读和维护。

7.4.1 函数定义的一般形式

1. 无参函数的定义形式

类型标识符　函数名()
{声明部分
语句
}

其中的“类型标识符”指明了函数返回值的类型。函数名是由用户自己定义，后面是空括号，代表没有函数参数，即代表无参函数，但是空括号不可以省略。

花括号中的内容被称为函数体。注意，在函数体中声明的各种对象，只能在函数体内有效。

一般情况下，无参函数没有返回值，因此可以将函数的类型标识符写成“void”。

举例：

```
void print()
{
    printf(" I am a boy. \n ");
}
```

可以从这个例子中看出，print 为函数名，它是一个无参函数，并且它没有返回值。当它被调用时，它的功能就是输出“I am a boy.”字符串。

2. 有参函数的定义形式

类型标识符 函数名（形式参数列表）
{声明部分
 语句
}

可以看出，有参函数与无参函数的主要区别就在于多了形式参数列表，在该列表中列出的形参被称为形式参数，简称为形参，它们可以是各种类型的数据，分别需要类型声明，之间用“,”分隔。函数被调用时，主调函数将通过实际参数（简称实参）传递实际的值给这些形参。

举例：定义一个有参函数，用来求两个数中较大的那个数，就可以写成

```
int max(int a,int b)
{
    if(a > b)return a;
    else return b;
}
```

在这个程序中，将 max 函数的返回类型定义为 int 型，a，b 为函数的形参，并且都声明

为 int 类型。当 max 被调用时，主调函数将通过实参将实际的值传给形参 a 和 b。函数体中的 if 语句用来判断，如果 a > b，使用 return 语句返回 a 的值，否则就返回 b 的值。

7.4.2　函数的参数和函数的值

通过前面的例子，了解到函数的参数分为形参和实参。那么这两种参数的特点和关系究竟是怎么样的呢？形参只是出现在函数定义中，在该函数体中可以被使用，但在函数体外就不能使用；实参只是出现在主调函数中，在调用函数时，把实参的值传递给被调函数的形参，从而实现主调函数向被调函数的数值传递，如图 7-1 所示。

函数的形参和实参具有以下几个特点：

1）形参只有在调用的时候才被临时分配内存单元，在调用结束后，就释放它所占有的内存空间。

2）实参不论是什么类型的数据，但必须有一个最终的值，在进行函数调用时，它都必须是有确定的值，从而把这些值传递给形参。

3）实参和形参的数据类型必须一致，否则程序在编译时会出错。

4）只能将实参的值传递给形参，反过来则不可以。

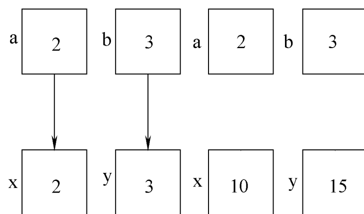


图 7-1　数值传递示意图

【例 7.8】

```
main ( )
{
    int num;
    printf("input a number\n");
    scanf("%d",&num);
    m(num);
    printf("%d\n",num);
}
int m(int n)
{
    int i;
    n = n * n;
    printf("%d\n",n);
}
```

在该例中，定义了一个 m 函数，该函数的功能是求 n 的平方值。在主函数中，输入 num 的值，然后调用 m 函数，num 作为实参，将值传递给形参 n。在这里，也要提醒大家注意，形参和实参的名字可以是一样，但是它们两个是完全独立的个体。在函数 m 中，形参 n 的值被改变，即 n = n * n；同时，函数体中的 printf 语句将 n 的值打印出来，当函数 m 返回主函数后，主函数中的 printf 语句又将 num 值打印出来，你可能会想到这里的值和刚才函数 m 中打印值一样，但要注意，在 C 语言中，形参是不能改变实参变量的值，因此，主函数中的 printf 语句所打印的 num 值仍为最开始输入的值。

7.4.3　函数的返回值

函数的返回值，指的是函数被调用后，返回主调函数的一个值。例如在前面例子中讲到的 max 函数，如果调用 max(2,5)，则返回值为 5，若为 max(7,3)，则返回值为 7。返回函数值时，应注意以下几点：

1）使用 return 语句返回函数值。

return 语句的一般形式为：

return 表达式；

或者

return（表达式）；

其中的“表达式”为函数返回主调函数的值，注意必须和函数的类型标识符一致。

2）如果在函数定义时没有写明类型标识符，则默认为整型。

3）如果函数不需要返回值，函数类型标识符则可以写作“void”，表示该函数没有返回值。

7.4.4　函数的调用

在 C 语言中，函数调用的一般形式为：

函数名（实际参数表）

其中的“实际参数表”，即实参表是可以没有的，表示无参函数。实参表可以是任何类型的数据，可以是常量、变量及表达式。各参数之间用“,”分隔。

C 语言的函数是可以相互调用的，但在调用函数前，必须对函数的类型进行声明，就算是标准库函数也不例外。标准库函数的声明会被按功能分别写在不同的头文件中，使用时只要在文件最前面用#include 预处理语句引入相应的头文件。例如前面经常使用的 printf 函数，其声明就是放在文件名为 stdio. h 的头文件中。调用就是指一个函数体中引用另一个已定义的函数来实现所需要的功能，这时函数体称为主调用函数，函数体中所引用的函数称为被调用函数。一个函数体中可以调用数个其他的函数，这些被调用的函数同样也可以调用其他函数，函数也可以自己调用自己。在 C 语言中，只有一个函数是不可以被其他函数调用的，那就是 main 主函数。

可以由以下几种方式调用函数：

1）函数表达式：例如，m = max（x，y）是一个赋值表达式，把函数 max 的返回值赋予变量 m。

2）函数语句：例如，printf（“%d”，m）；即直接写上函数名加上分号。

3）函数实参：例如，printf（“%d”，max（x，y））；即将函数作为另一个函数调用时的实际参数。该语句的作用是将 max 函数的返回值作为 printf 函数的实参。

7.4.5　被调用函数的声明和函数原型

函数在被调用之前，需要在主调函数中对其进行类型声明，类似于在变量使用之前，需要进行定义声明一样。这样做的目的是为了使编译系统能知道被调函数返回值的类型，以便在主调函数中对返回值做相应的处理。

其一般形式为：

类型声明符　被调函数名（类型 形参，类型 形参…）；

或者

类型声明符　被调函数名（类型，类型…）；

括号内给出了形参的类型和形参名，或只给出形参类型。这样做的目的是便于编译系统进行检错，以防止可能出现的错误。

例如，可以在主调函数中对 max 函数的声明为：

```
int max(int x,int y);
```

或者

```
int max(int,int);
```

并不是所有的函数在主调函数中都需要进行声明，以下几种情况可以省去主调函数中对被调函数的函数声明：

1）如果函数的返回值类型为字符型或者整型的，则可以不用声明。当遇到字符型时，C 编译系统会自动将其转换成整型处理。

2）在函数定义之前，在主函数外部对其进行声明了，则不需要在后面的主调函数中再对其进行声明。例如：

```
float area(float x,float y);
char name(char n);
main()
  {
         ⋮
        area(x,y);/* 无需再进行声明 */
        name(n);/* 无需再进行声明 */
  }
  float area(float a,float b)
  {
         ⋮
  }
char name(char m)
  {
         ⋮
  }
```

3）如果被调函数的定义的位置出现在主调函数之前时，这时也可以不对被调函数做声明而直接调用。

4）对系统库函数的调用则无需再加声明，但必须使用 include 命令，把函数相应的头文件包含于源文件前部。

7.4.6　函数的嵌套调用

函数的嵌套调用与其他语言的子程序嵌套调用的情形是类似的。在执行被调用函数时，被调用函数又调用了其他函数。其关系可表示如图 7-2 所示。

该图表示了双层函数嵌套调用的情况：main 函数中调用 a 函数时，在执行 a 函数过程

中，a 函数调用了 b 函数，b 函数执行完毕后再返回 a 函数的转出点继续执行剩余代码，a 函数执行完毕后再返回 main 函数的转出点继续执行剩余代码。

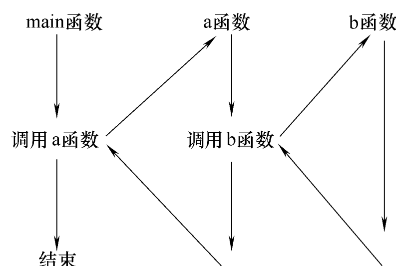


图 7-2　函数的嵌套调用

【例 7.9】　函数的嵌套调用。

```
main()
{
a();/*在主函数中调用a()函数*/
}
a()
{
b();/*在a()函数中调用b()函数*/
printf("欢迎使用本程序！\n");
}
b()
{
printf("*****************\n");
}
```

程序执行结果为：

```
*****************
欢迎使用本程序！
```

7.4.7　函数的递归调用

一个函数在它的函数体内，直接或间接地调用它自身，就称为函数的递归调用。C 语言是允许递归调用的。在递归调用中，函数既是主调函数，又是被调函数，反复调用自身，每调用一次就进入新的一层。但是为了避免无休止地调用自身的情况，因此一定要给递归函数中设置一个终止调用的手段，通常的办法是使用条件判断语句，一旦条件满足后就终止递归调用，一层一层地返回。

例如有这样一个程序：

```
int a(int x)
{
  int y;
  z=a(y);
  return z;
}
```

可以看出，这是一个递归调用程序，a 函数在函数体中调用了自身，但是也不难发现，这是一个死循环函数，程序将陷入无休止的递归调用过程，这并不是所期望的。因此将这个程序加入一些判断语句：

```
int a (int n, int x)
{
```

```
    int y;
    if (n > 0)
      z = a (n-1, y);
    else
      return z;
  }
```

现在，在程序中加入了一个变量 n，这个变量就是用来控制递归调用的。当 n >0 时，函数继续调用自身，一旦 n< =0 就终止递归调用，然后逐层返回。

再来看一个例子：

【例 7.10】 计算 x 的 n 次方（x 和 n 均为正整数）。

```
main()
{
int a,y;
scanf("%d,%d",&a,&y);
printf("%d**%d=%d\n",a,y,power(a,y));
}
power(int x,int n)
{
int p;
if(n>0)
p=power(x,n-1)*x;
else
p=1;
return(p);
}
```

在该例中，power 函数通过了 power（x，n-1）直接调用了它自身，通过变量 n 来控制了程序调用的次数，从而实现了计算 x 的 n 次方。

递归调用的优点是使程序看上去简洁明了，可以使一些原本复杂的程序简化，但是由于每次调用一个函数时都需要存储空间来保存调用“现场”，以便后面返回，并且递归调用往往涉及同一个函数的反复调用，所以它要占用很大的存储空间，特别是在递归调用次数较多的情况下，将导致程序运行速度较慢。

7.4.8　数组作为函数参数

我们已经知道可以用任何类型的变量来作为函数的参数，数组同样也可以作为函数参数。用数组作函数参数有两种形式，一种是把数组元素作为实参使用；另一种是把数组名作为函数的形参或实参使用。

1. 数组元素作函数实参

这种形式与普通变量一样，因此它作为函数实参时的使用方法与普通变量是相同的，在调用函数时，把作为实参的数组元素的值传送给形参，实现单向的值传送。

【例 7.11】 写一函数，统计字符串中字母的个数。

```
    int   isalp( char c)
      { if   (c > ='a'&&c < ='z'||c > ='A'&&c < ='Z')
              return(1);
          else   return(0);
      }
  main( )
    { int  i,num =0;
          char str[255];
          printf("Input   a   string:");
          gets(str);
          for(i =0;str[i]!  =‘\0’;i + +)
          if(isalp(str[i]))      num + +;/* 数组元素的值作为实参传递给形参*/
          puts(str);
          printf("num =%d\n",num);
          getch();
    }
```

在该例中首先定义一个整型函数 isalp（），并声明其形参 c 为字符型变量，在其函数体中根据 if 语句判断输出相应的结果。在 main 函数中用一个 gets 语句输入字符给 str，然后通过循环语句调用 isalp 函数，将其返回值用来统计字符串中字母的个数。

说明：

1）用数组元素作实参时，只要求数组类型和函数的形参类型一致即可，并不要求函数的形参也是下标变量。换句话说，对数组元素的处理是按普通变量对待的。

2）在普通变量或下标变量作函数参数时，形参变量和实参变量是由编译系统分配的两个不同的内存单元。在函数调用时发生的值传送，是把实参变量的值赋予形参变量。

2. 数组名作为函数参数

用数组名与用数组元素作函数实参有以下不同：

1）用数组元素作函数参数时的处理是按普通变量对待的，但用数组名作函数参数时，则要求形参和相对应的实参都必须是类型相同的数组，都必须有明确的数组声明。当形参和实参两者不一致时，即会发生错误。

2）在普通变量或数组元素作为函数参数时，变量是由编译器分配的两个不同的内存单元。在函数调用时，实参变量的值将赋予形参变量。在用数组名作为函数参数时，不是进行值的传送，并不是把实参数组的每一个元素的值都赋予形参数组的各个元素。因为实际上形参数组并不存在，编译器不会为形参数组分配内存。看到这里，或许你会问，那么数据到底是如何传送的呢？我们曾介绍过，数组名就是数组的首地址。数组名作函数参数时，其实是将地址进行了传送，也就是说把实参数组的首地址赋予形参数组名。因此，形参数组和实参数组其实就是同一个数组，它们共同占用一段内存空间。

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
起始地址 2000	2	4	6	8	10	12	14	16	18	20
	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]

图 7-3　数组元素分布

从图7-3中，可以直观地看出它们之间的关系。设a为整型数组，数组a的起始地址为2000，由此开始连续若干内存空间被a所占用。b为形参数组。当函数被调用时，进行地址传递，把实参数组a的首地址传送给形参数组名b，于是b的首地址便为2000。于是a、b两数组共占2000为首地址的一段连续内存单元。从图中还可以看出a和b下标相同的元素实际上也占相同的两个内存单元（整型数组每个元素占2个字节）。例如，a［1］和b［1］都占用2002和2003单元，当然a［1］等于b［1］。类推则有a［i］等于b［i］。

【例7.12】 有1个一维数组score，内放10个学生成绩，求平均成绩。

```
float average(float array[10])
{int i;
 float aver,sum = array[0];
 for(i = 1;i < 10;i + + )
 sum = sum + array[i];
 aver = sum/10;
 return(aver);
}
void main(void)
{ float score[10],aver;
 int i;
 printf("input 10 scores:\n");
 for(i = 0;i < 10;i + + )
 scanf("%f",&score[i]);
 printf("  \n");
 aver = average(score);
 printf("average score is %5.2f",aver);
 }
```

运行情况如下：

```
input 10 scores:
56 78 98.5 76 87 99 67.5 75 97 ↙

average score is 83.40
```

3）变量作函数参数时，传送的值是单向的，即只能从实参传向形参，而不能从形参传回实参。形参的初值和实参相同，如果程序中形参的值发生了改变，但实参值仍保持不变。而当用数组名作函数参数时就不一样了。由于形参和实参共享一组内存空间，因此当形参数组发生变化时，实参数组也随之变化。

7.5　局部变量和全局变量

之前使用了各种变量，但是任何一个变量都是有它的管辖范围的（作用域），就是说，一个变量在定义了之后，并不是在程序的任何地方都可以使用这个变量。只有在变量的作用域内才能使用这个变量。在C语言中，如果按作用域分，变量分为局部变量和全局变量。

7.5.1 局部变量

在一个函数内部定义的变量被称作局部变量，这种变量的作用域是在本函数范围内。通俗一点说，局部变量只能在定义它的函数内部使用，而不能在其他函数内使用这个变量。

```
float a(int n)
{
    int x,y;
}
```

变量 n、x、y 只有在函数 a 中有效。

```
main ( )
{
      int m,n;
}
```

变量 m、n 在函数 main 内部有效。

说明：

1）main 函数作为主函数，也是一个函数，它内部定义的变量也只能在 main 函数内部使用，而不能在其他函数内部使用 main 函数内部定义的变量。

2）不同的函数中可以使用相同的变量名，但它们是不同的变量。记得函数在执行时，系统要给它分配一块单独的内存吗？所以虽然变量名是相同的，但系统看到它们定义在不同的函数中，就认为它们是不同的变量。这样做有个好处，可以在函数内部，根据需要设置任何的变量名。如果不是这样，那么在一个函数内定义了一个变量名后，在其他函数内就不能再使用相同的变量名了。

3）形参也属于局部变量，作用范围在定义它的函数内。所以在定义形参和函数体内的变量时是肯定不能重名的。

4）在复合语句内部也可以定义变量，这些变量的作用域只在本复合语句内。只在需要的时候再定义变量，这样做可以提高内存的利用率。

```
f(int m)
{
    int n;
    {int s;s=m+n;}/*变量 s 只在此复合语句内起作用*/
}
```

【例 7.13】

```
main ( )
{
    int i=5, j=5, k;
    k=i+j;
    {
        int k=12;
        printf ("%d\n", k);
    }
```

```
    printf ("%d\n", k);
}
```

例 7.13 在 main 函数中定义了 i、j、k 3 个变量，其中变量 k 未赋初值。而在复合语句内又定义了一个变量 k，并赋初值为 12。注意这两个 k 是两个变量。在复合语句外程序部分，由 main 函数中定义的 k 起作用，而在复合语句内则由在复合语句内定义的 k 起作用。因此程序第 4 行的 k 为 main 函数所定义，其值应为 10。第 7 行输出 k 值，因为它在复合语句内，由复合语句内定义的 k 起作用，其初值为 12，所以输出值为 12，第 9 行输出 k 值，因为在复合语句之外，输出的 k 应为 main 函数所定义的 k，所以 k 值由第 4 行已获得为 10，故输出为 10。

7.5.2　全局变量

全局变量也称为外部变量，它是在函数外部定义的变量。它不属于哪一个函数，它属于一个源程序文件，全局变量可以为本文件中其他函数所共用，它的有效范围为从定义变量的位置开始到本源文件结束。

例如：

```
int a, b;               /*外部变量 a、b*/
void f1 ()              /*函数 f1*/
{
    ⋮
}
float x, y;             /*外部变量 x、y*/
int f2 ()               /*函数 f2*/
{
    ⋮
}
main ()                 /*主函数*/
{
    ⋮
}
```

可以从该例中看出 a、b、x、y 都是在函数外部定义的外部变量，所以是全局变量。但应注意到 x、y 定义在函数 f1 的后面，f1 函数内没有对 x、y 的声明语句，所以 x、y 在 f1 内是无法使用的。a、b 定义在源程序首行，因此在 f1、f2、main 函数中无需加入声明语句。

【例 7.14】　外部变量与局部变量同名时的情况。

```
int a=5,b=7;            /*a、b 为外部变量*/
max(int a,int b)        /*a、b 为外部变量*/
{int c;
 c=a>b? a:b;
 return(c);
}
main()
```

```
{int a =10;
 printf("%d\n",max(a,b));
}
```

运行结果：10

说明：

1）全局变量的作用是为了增加函数间数据联系的渠道；

2）建议不在必要时不要使用全局变量，因为：

① 全局变量在程序的全部执行过程中都占用存储单元，而不是仅在需要时才开辟单元；

② 它使函数的通用性降低了；

③ 使用全局变量过多，会降低程序的清晰性，人们往往难以清楚地判断出瞬时各个外部变量的值。

3）如果在同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量被“屏蔽”，它不起作用。

第 8 章　指针、结构体与共用体

8.1　指针和地址

指针是 C 语言中一个非常重要的概念，也是 C 语言的一个重要特色。指针的概念比较复杂，使用也比较灵活，每一个学习和使用 C 语言的人，都应当深入地学习和掌握指针，没有掌握指针就没有掌握 C 语言的精华。

1. 指针的定义

指针，又称为指针变量，是用于存放其他变量的地址。指针变量所包含的是内存地址，而该地址便是另一个变量在内存中存储的位置。指针它本身也是一种变量，和其他变量一样，要占有一定数量的存储空间，用来存放指针值（即地址）。

2. 指针的作用

C 语言中引入指针类型变量，不仅简单地实现了类似于机器间接寻址操作的指令，更重要的是使用指针可以实现程序设计中利用其他数据类型很难实现，甚至无法实现的工作。指针的作用主要体现在以下几点：

1）使代码更为紧凑和有效，提高了程序的效率；

2）便于函数修改其调用参数，为函数间各类数据传递提供简捷而便利的方法；

3）实现动态分配存储空间。

正确地使用指针会带来许多好处，但是指针操作也有难以掌握的一面，若使用不当，可能会产生程序失控甚至造成系统崩溃。

8.2　指针变量和指针运算符

1. 指针变量的定义

指针变量是存放地址的变量，和其他变量一样，必须在使用之前，要对其进行声明。其定义声明的一般形式为：

类型声明符 * 指针变量名；

举例：

int * point_1；　　指针变量 point_1 是指向 int 类型变量的指针。

char * str；　　指针变量 str 是指向 char 类型变量的指针。

float * f1；　　指针变量 f1 是指向 float 类型变量的指针。

static int point_2；　　指针变量 point_2 是指向 int 类型变量的指针，并且该指针本身的存储类型是静态变量。

int * pf_1( )；　　pf_1（ ）是一个函数，该函数返回指向 int 类型变量的指针。

int(* pf_2)( )；　　pf_2 是指向一个函数的指针，该函数返回 int 类型变量。

注意，指针变量的值表达的是某个数据对象的地址，只允许取正的整数值。然而，不能

因此将它与整数类型变量相混淆。所有合法指针变量应当是非 0 值。如果某指针变量取 0 值，即为 NULL，则表示该指针变量所指向的对象不存在。这是 NULL 在 C 语言中又一特殊用处。

2. 指针运算符 & 和 ∗

指针变量中只能存放地址，它最基本的运算符是 & 和 ∗ 。

（1） & —— 取地址运算符

它的作用是返回变量（操作数）的内存地址。注意，它只能用于一个具体的变量或数组元素，不可用于表达式。

举例：

```
int * p_1;
int n;
n = 500;
p_1 = &n;
```

这段程序是将整型变量 n 的地址赋值给指针变量 p_1。

（2） ∗ —— 指针运算符，或“间接访问”运算符

其作用是返回地址（指针变量所表达的地址值）中的变量值。

举例：

```
int * p_2;
int m;
int n;
m = 500;
p_2 = &m;
n = * p_2;
```

这段程序是将整型变量 m 的地址赋值给指针变量 p_2，然后通过使用“ ∗ p_2”间接取了变量 m 的值，将其赋值给 n，其作用与 m = n 是一样的，但是运用了指针变量。

“&”与“ ∗ ”运算符互为逆运算。

例如，对指针变量 p_3 指向的目标变量 ∗ p_3 实行取址运算：&（ ∗ p_3）。其结果就是 p_3。

对变量 a 的地址实行取值运算： ∗ (&a)。其结果就是 a。

注意，a 与 ∗ p_3 同级别可写成 a = ∗ p_3 或 ∗ p_3 = a。p_3 与 &a 同级别可写成 p_3 = &a，但是决不可以写成 &a = p_3，也不能写成 p_3 = a。

3. 指针的初始化和赋值运算

指针的初始化往往与指针的定义声明同时完成，它的一般格式为：

类型声明符 ∗ 指针变量名 = 初始地址值;

举例：

```
char c1;
char * cp = &c1;
```

注意：

1）任何一个指针在使用之前，必须加以定义声明，必须经过初始化。未经过初始化的指针变量禁止使用。

2）在声明语句中初始化，也是把初始地址值赋给指针变量，只有在声明语句中，才允许这样写。而在赋值语句中，变量的地址只能赋给指针变量本身。

3）指针变量初始化时，变量应当在前面声明过，这样才可以使用“& 变量名”作为指针变量的初始值。否则，编译时将出错。

4）必须以同类型数据的地址来进行指针初始化，即赋值运算操作仅限制在同类之间才可以实现。

【例 8.1】 指针变量的引用。

```
main()
{
int x = 100
int * p_1 = &x, * p_2;
/ *  p_1 指向 x * /
p_2 = p_1;
/ *  p_2 指向 x * /
printf("%d\n", * p_2);
}
```

执行结果：100

【例 8.2】 指针变量的运算。

```
main()
{
  char c = 'A';/ * 变量 c 的初始值为'A' * /
  char * charp = &c;/ * 变量 c 的地址作为指针变量 charp 的初始值 * /
  printf("%c%c\n",c, * charp);
  c = 'B';/ * 变量 c 赋值为‘B’ * /
  printf("%c%c\n",c, * charp);
   * charp = 'a';/ * 将指针变量 charp 所指向地址的内容改为'a' * /
  printf("%c%c\n",c, * charp);
}
```

执行结果应显示：

AA

BB

Aa

4. sizeof 运算符

使用 sizeof 运算符可以准确地得到在当前所使用的系统中某一数据类型所占字节数。它的格式为：

sizeof（类型声明符）

其运算值为该类型变量所占字节数。用输出语句可方便地打印出有关信息，例如：

```
printf("%d\n",sizeof(int));
printf("%d\n",sizeof(float));
printf("%d\n",sizeof(char));
```

5. 指针的算术运算

所谓指针的算术运算，就是指按地址计算规则进行。由于地址计算与相应数据类型所占字节数有关，所以指针的算术运算应考虑到指针所指向的数据类型。

指针的算术运算只有 4 种：+，-，++，--。

(1) 指针自增 1 运算符“++”和指针自减 1 运算符“--”：

指针的自增 1 运算是指指针向后移动一个数据的位置，所谓一个数据的位置就是指一个数据类型所占的字节数，即指向的地址为原来地址 + sizeof（类型声明符）。

例如：

```
int a, *p;
p = &a;
p++;
```

这段程序的结果是，指针变量 p 指向了变量 a 的地址 + sizeof（a）的位置。

(2) 指针变量的“+”和“-”运算

指针变量 p 加（+）正整数 n，表示指针向后移过 n 个数据类型，使该指针所指向的地址变为原指向的地址 + sizeof（类型声明符）*n。指针变量减（-）正整数 n，表示指针往前移回 n 个数据，使该指针所指向的地址变为原指向的地址 - sizeof（类型声明符）*n。

6. 指针的关系运算

指针也可以进行关系运算，只有指向同一种数据类型的指针，才有可能进行各种关系运算。指针的关系运算符包括：==，!=，<，<=，>，>=。

指针的关系运算实际是指两个指针所指向的地址进行比较。两个不同类型的指针进行比较是没有意义的，与常量、变量比较也没有意义。但是有一种情况是例外的，指针常常与常量 0 进行比较，用来判断是否为空指针，例如有一个指针 p，就可以使用 p==0 或 p!=0 来判断其是否为空指针。

7. 使用指针编程中的常见错误

(1) 使用未初始化的指针变量。例如：

```
main()
{
    int a, *p1;
    a = 0;
    *p1 = a;    /* 指针变量 p1 未初始化 */
      ⋮
}
```

(2) 指针变量所指向的数据类型与其定义的类型不符。例如：

```
main()
{
    float x, y;
    short int *p2;
    p2 = &x;    /* 变量 x 与指针变量 p2 数据类型不符 */
    y = *p2;    /* 变量 y 与指针变量 p2 数据类型不符 */
        ⋮
```

```
}
```

(3) 指针的错误赋值。例如：

```
main ( )
{
    int x, * p3;
    x = 10;
    p3 = x;   /* 错误的赋值方式 */
      ⋮
}
```

(4) 用局部变量的地址去初始化静态的指针变量。例如：

```
int glo_ a;
func ( )
{
    int loc_1, 1oc_2;
    static int * glo_ p = &glo_ a;
    static int * 1oc_p = &1oc_1; /* 用局部变量 loc_1 的地址去初始化静态的指针 loc_p */
    int * p = &1oc_2;
      ⋮
}
```

8.3　指针与函数参数

指针在函数调用中，可以作为参数，以此来改变主调函数中变量的值。我们都知道一般的变量作函数参数时，其值的传递都是单向的，但是指针作参数时却不是，让我们来看一个例子。

【例 8.3】　x 和 y 的值互换。

```
swap1(x,y)
int x,y;   /* 局部变量不能带出函数 */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
main( )
{
    int a,b;
    a = 10;
    b = 20;
    swap1(a,b);
```

```
    printf("a = %d b = %d\n",a,b);
}
```

执行结果：

a = 10 b = 20

从该程序可以看出，被调函数不能影响主调函数中变量的值。虽然，swap1（）函数中，x 和 y 的值互相交换了，但 x、y 是局部变量，主函数对该函数的调用 swap1（a，b）是传值调用，x 与 y 的交换结果，不能影响调用者中的变量 a 和 b，因此 a 与 b 的值并未交换。

现在换一种方式，使用指针作为函数参数。

【例 8.4】 x 和 y 值互换。

```
swap2 (px, py)
int *px, *py;       /* 地址变量 */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;     /* 返回后地址变量释放 */
}
main ()
{
    int a, b;
    a = 10;
    b = 20;
    swap2 (&a, &b);     /* 传地址 */
    printf (" a = %d b = %d \n", a, b);     /* 得到交换的值 */
}
```

执行结果：

a = 20 b = 10

从该程序可以看出，使用了指针作为函数的参数。a 与 b 的地址，即 &a 与 &b 并未交换，但是它们的值已经被交换了。

8.4　指针、数组和字符串指针

1. 指针和数组

用指针和用数组名访问内存的方式几乎完全一样，但是又有着微妙而重要的差别，即指针是地址变量；数组名则是固定的某个地址，是常量。

现在来举一个简单的例子：

若定义一个数组：int a [10]；

那么这个数组 a 则在内存中占有从一个基地址开始的一块足够大的连续的空间，用来容下 a [0]，a [1]，…，a [9]。基地址（或称为首地址）是 a [0] 存放的地址。其他依下标顺序排列。

若定义一个指针：int * p； p = a；

则表示 p = &a［0］；即 p 指向 a［0］，p + 1 指向 a［1］，p + 2 指向 a［2］，以此类推。

说明：在 C 语言中，数组名本身也是指向 0 号元素的地址常量。因此，p = &a［0］可写成 p = a，a 当作常量指针，它指向 a［0］。p = a + i 与 p = &a［i］等价，故数组名可当指针用，它指向 0 号元素。p + i 指向 a［i］，a + i 也指向 a［i］，这样，a［i］与 *（p + i）或 *（a + i）可看成是等价的，可互相替代使用。

注意，由于 a 只是数组名，是地址常量，而不是指针变量，所以可以写 p = a，但不能写 a = p，可以用 p + +，但不能用 a + +。

2. 字符串指针

我们知道使用 char 类型的数据是用来处理字符串的，而数组又可用相应数据类型的指针来处理，所以可以用 char 型指针来处理字符串。通常把 char 型指针称为字符串指针或字符指针。

我们来看一个例子：

【例 8.5】 strlen（）是使用字符串指针来计算字符串长度的函数。

```
strlen(s)
char *s;
{
    int n;
    for(n =0; *s! ='\0';s + + )
    n + +
    return(n);
}
main()
{
    static char str[] = {"Program"};
    printf("%c\n", *str);
    printf("%d\n",strlen(str));
    printf("%c\n", *(str+1));
}
```

执行结果：

```
P
7
r
```

定义字符指针可以直接用字符串作为初始值，来实现初始化，可以将程序中的 char str［］ = {" Program"}；改写成：

```
char *str = "Program";
```

或者

```
char *str;
str = "Program";
```

注意，这样的赋值并不是将字符串复制到指针中，只是使字符指针指向字符串的首地

址。但对于数组，例如：

```
char str[10];
str = "Program";
```

这样写是不对的，str 是数组名，而不是指针，只能按字符数组初始化操作，即

```
static char str[] = {"Program"};
```

当字符串常量作为参数（实参）传递给函数时，实际传递的是指向该字符串的指针，并未进行字符串复制。

【例 8.6】 向字符指针赋字符串。

```
main ()
{
      char s = "Hello!";
/ * 相当于 s 数组 * /
      char * p;
      while( * s!  = '\0')
      printf( "%c" , * s + + ) ;
      printf( " \n" ) ;
      p = "Good-bye!" ;
/ * 指针指向字符串 * /
      while( * p!  = '\o')
      printf( "%c" , * p + + ) ;
      printf( " \n" ) ;
}
```

执行结果：

Hello!

Good-bye!

在这个例子中字符串是逐个字符输出的，也可以字符指针为变量，作字符串输出，例如，

```
printf( "%s" ,s) ;
```

或

```
printf( "%s" ,p) ;
```

【例 8.7】 以字符指针为参数来调用字符串比较函数 strcomp（）。

```
strcomp( s,t)
char  * s, * t;   / * 形式参数为字符指针 * /
{
     for( ; * s = = * t;s + + ,t + + )
     if( * s = = '\0')
     return(0) ;
     return( * s - * t) ;   / *  字符串相同返回零值 * /
}
main( )
```

```
{
    static char s1[ ] = {"Program"};
    char * s2 = "Hello!";
    printf("%d\n",strcomp(s1,s2));
    printf("%d\n",strcomp("Hello",s2));
}
```

strcomp（）函数的形参 s 和 t 初值已由实参传递过来了，所以 for 语句的第一个表达式可以省略不写。

用指针处理字符串的复制，比用数组处理更精练。例如：

```
strcpy(s,t)
char *s, *t;
{
    while( *s++ = *t++);
}
```

如果用数组处理则要复杂些，例如：

```
strcpy(s,t)
char s[ ],t[ ];
{
    int i;
    i=0;
    while((s[i]=t[i])! ='\0')
    i++;
}
```

直接演化出指针处理，即

```
strcpy (s, t)
char *s, *t;
{
    while ( (*s= *t)! ='\0')
    {
        s++;
        t++;
    }
}
```

第 1 段程序是第 3 段程序的进一步简化，而且不必进行与'\0'的比较。

8.5　指针数组

1. 指针数组的定义和声明

同一类型的指针变量的集合，称为指针数组。换句话说，一个数组里面的元素是指针变量，就称为指针数组。这些指针变量都为相同的存储类型，并且指向的目标数据类型也是相

同的。

指针数组的一般表示格式为：

类型声明符 * 指针数组名[元素个数]；

例如：

float * p_1[10]；

p_1[10]是一个指针数组，含有 10 个指针，并指向 float 型的数据。

2. 指针数组的初始化

指针数组的初始化可以与定义声明同时进行。与一般数组一样，只有全局的或静态的指针数组才可进行初始化，另外，不能用局部变量的地址去初始化静态指针。

【例 8.8】 指针数组。

```
main()
{
  static int b[2][3] = {{1,2,3},{4,5,6}};
  static int * pb[] = {b[0],b[1]};
/ * 指针数组指向行首 * /
  int i,j;
  for (i =0;i <2;i + +){
     for(j =0;j <3;j + +)
           printf("b[%d][%d] =%d",i,j, * (pb[i] +j));
     printf("\n")
  }
}
```

执行结果：

b[0][0] =1 b[0][1] =2 b[0][2] =3

b[1][0] =4 b[1][1] =5 b[1][2] =6

在这个例子中，我们将一个二维数组 b［2］［3］分解成两个一维数组。它们的首地址，分别为 b［0］和 b［1］，也可以理解为第 1 行的首地址和第 2 行的首地址，并被赋给指针 pb［0］和 pb［1］。

3. 字符指针数组

字符指针可以用来处理一个字符串，那么字符指针数组则可以用来处理多个字符串。

【例 8.9】 用字符指针数组处理多个字符串。

```
main()
{
      void sort();
      void print();
      static char * name[ ] = {"Follow me","BASIC","Great Wall","FORTRAN","Com-
      puter design"};/ * 指针数组指向各字符串 * /
      int n =5;
      sort(name,n);
      print(name,n);
```

```
}
void sort(name,n)
char * name[ ];int n;
{
      char * temp;
      int i,j,k;
      for(i =0;i < n;i + +)
      {
        k =i;
        for(j =i +1;j + +;j < n)/ * 指向各字符串的比较 * /
        if(name[j] < name[i])k =j;
        if(k!  =i)
        {temp = name[i];name[i] = name[k];name[k] = temp;}
      }
}
void print(name,n)
char * name[ ];int n;
{
      int i;
      for(i =0;i < n;i + +)
            printf("%s\n",name[i]);/ * 按排好序的指向输出字符串 * /
}
```

运行结果为：

```
BASIC
Computer design
FORTRAN
Follow me
Great Wall
```

8.6　多级指针

所谓多级指针，指的就是指针的指针，也就出现了多级间接寻址访问的现象。多级指针也称为指针链，一般很少会有超过二级的指针，过多的级数会使程序维护困难，而且也容易出错。因此，常用的多级指针也只是二级指针，它的一般格式为：

类型声明符 * * 指针变量名；

例如：int * * p;

　　static char * * charp;

【例 8.10】　二级指针的应用。

```
main()
{
```

```
    int x, * p, * * q;
    x = 10;
    p = &x;
    q = &p;
    printf("%d\n", * * q);
}
```

在实际使用中,二级指针常用来处理字符指针数组。

【例 8.11】

```
#include
main()
{
    char * * pp;
    static char * di[] = {"up","down","left","right",""};/* 字符指针
    数组指向字符串 */
    pp = di;/* 二级指针 pp 指向字符指针数组 di */
    while( * * pp! = NULL)
    printf("%s\n", * pp + + );
}
```

执行结果:

```
up
down
left
right
```

【例 8.12】 指针数组指向整型数组。

```
main()
{
  static a[5] = {1,3,5,7,9};/* 整型数组 */
  static int * num[5] = {&a[0],&a[1],&a[2],&a[3],&a[4]};/* 指针数组赋值 */
  int * * p ,i;
  p = num;/* 指向指针数组 */
  for(i = 0;i < 5;i + + )
     {printf("%d\t", * * p);p + + ;}
}
```

运行结果:

1 3 5 7 9

8.7 返回指针的函数

一个函数被调用后，可以返回各种类型的数据，同样也可以返回指针数据，也就地址。

【例 8.13】　有若干学生，每个学生 4 门课，要求输入学生序号后，能输出该学生的全部成绩。

```
main()
{
  static float score[][4] = {{60,70,80,90},{56,89,67,88},{34,78,90,66}};
float * search();/* 声明函数返回指针 */
  float * p;
  int i,m;
  printf("enter the number of student:");
  scanf("%d",&m);
  printf("The scores of No. %d are:\n",m);
  p = search(score,m);/* 得到该位同学的行首地址,指向列 */
  for(i = 0;i < 4;i + +)
  printf("%5.2f\t", *(p + i));
}
float * search(pointer,n)
float( * pointer)[4];
int n;
{float *  pt;
  pt = *(pointer + n);/* 指向列 */
  return(pt);
}
```

运行结果：

```
enter the number of student:1 ↙
The score of No. 1 are:
56.00 89.00 67.00 88.00
```

8.8　函数指针

1. 函数指针的定义和声明

函数与数组拥有类似的特性，可以用函数名表示函数的存储首地址，即执行该函数的入口地址。指向函数入口地址的指针，就称为函数指针。函数指针定义声明的格式如下：

数据类型声明(* 函数指针名) ( );

例如：int(* fp) ( );

2. 函数指针的作用

例如上例中的 fp 函数指针，运用实现访问目标的运算符“ * ”，即 (* func) ( )，它的作用是使程序控制转移到指针所指向的地址去执行该函数。所以，函数指针所指向的是程序代码区，而不是数据区。这与一般数据指针变量有原则的区别，一般数据指针指向数据区。

8.9　结构与联合

8.9.1　结构的定义

1. 结构的定义及其一般格式

我们知道，数组是将相同类型的元素组成单个逻辑整体的一种数据类型。而结构则是将不同类型元素组成单个逻辑整体的一种数据类型。结构是 C 语言中一种强有力的且特殊的数据类型，也是很重要的概念之一。

结构定义的一般格式如下：

```
struct [结构类型名]
{
        类型声明符 成员变量名;
           ⋮
        类型声明符 成员变量名;
} [结构变量列表];
```

例如，日期是由年、月、日组成的：

```
int year;
int month;
int day;
```

但是这种表达方式并不能很好地表现它们彼此的关系，因此用下面的结构定义来表示：

```
struct date
{
    int year;
    int month;
    int day;
}
```

由此，可以看出，年、月、日是日期的组成部分，可以把它们当成一个整体来看待，它们是该结构的成员。结构的成员可以是各种不同的数据类型。例如：

```
struct person
{
    int age;
    char name[10];
}
```

上面两例定义了两个结构的数据，data 和 person。可以用这些已经定义好的结构名来定义其他相同性质的结构。例如：

```
struct date today,yesterday;/*定义了其他两个相同的结构体 today、yesterday*/
struct person man,woman;/*定义了其他两个相同的结构体 man、woman*/
```

也可以将上述例子写成：

```
struct date
```

```
{
    int year;
    int month;
    int day;
} today,yesterday;
```

或者

```
struct
{
    int year;
    int month;
    int day;
} today,yesterday;
```

或者

```
struct
{
    int year;
    int month;
    int day;
    }date today,yesterday;
```

这 3 种方式都是合法的，但是第 3 种写法的风格比较好。

2. 结构的存取

结构变量成员可作为单独变量来处理，也就是说，可以直接访问结构中的一个成员变量，通过对成员变量的存取来实现对结构变量的存取，其格式如下：

结构变量名 . 成员变量名

【例 8.14】　显示输入的年、月、日。

```
main ( )
{
    struct date /* 定义了结构类型 */
    {
        int month;
        int day;
        int year;
    };
    struct date today;   /* 结构体变量 today */
    printf("Enter today's date(年,月,日)\n");
    scanf("%d,%d,%d",&today. year,&today. month,&today. day);
    printf("Today's date is %d/%d%/d\n",today. year,today. month,today. day );
}
```

执行后可以显示出所输入的年、月、日。

【例 8.15】　对外部存储类型的结构体变量的初始化。

```
struct student /＊定义结构类型＊/
{
    long int num;
    char name[20];
    char sex;
    char addr[20];
}a = {10001,"Zhang San",'M',"1 Nanjing Road"};/＊结构体变量赋初值＊/
main()
{
    printf("No. :%ld\nname:%s\nsex:%c\naddress:%s\n",a.num,a.name,a.sex,
a.addr);
}
```

【例 8.16】　对静态存储类型的结构体变量的初始化。

```
main ()
{
    static struct student /＊静态结构类型＊/
    {
    long int num;
    char name[20];
    char sex;
    char addr[20];
}a = {10001,"Zhang San",'M',"1 Nanjing Road"};/＊赋初值＊/
printf("No. :%ld\nname:%s\nsex:%c\naddress:%s\n",a.num,a.name,a.sex,a.addr);
}
```

8.9.2　结构数组

同一类结构变量的集合也可以构成结构数组，例如：

```
struct person
{
    int age;
    char name[10];
};
static person persons[10];
```

这段程序定义了 10 个结构变量所组成的数组。

【例 8.17】　输入候选人名单,并对每个人计票,统计输出每个人的得票结果。

```
struct person /＊全局结构类型＊/
{
    char name[20];
    int count;
}leader[3] = {"Li",0,"Zhang",0,"Sun",0};/＊初始化＊/
```

```
main()
{
    int i,j;
    char leader_name[20];
    for(i=1;i<=10;i++)
    {scanf("%s",leader_name);/*输入人名*/
    for(j=0;j<3;j++)
    if(strcmp(leader_name,leader[j].name)==0)
    leader[j].count++;/*累加票数*/
    }
    printf("\n");
    for(i=0;i<3;i++)
    printf("%5s:%d\n",leader[i].name,leader[i].count);/*每人得票数*/
}
```

8.9.3　结构与函数

1. 向函数传递结构变量成员

结构变量成员也可以像普通变量一样，将值传递给函数。例如：

```
struct person
{
    int age;
    char name [10];
} worker;
```

这个结构中的任何一个成员变量均可以作为函数的参数，如：

```
f_1(worker.age);
f_2(worker.name);
```

2. 向函数传递完整的结构

不仅可以向函数传递结构变量成员，也可以向函数传递完整的结构。在传递完整的结构时，应当注意以下几点：

1）按传值方式传递整个结构。这种方式和普通变量一样，但是与数组不同，在函数内所引起结构参数中某个值的变化，只影响函数调用时所产生的结构复制，不影响原来的结构。

2）结构定义也是分为局部定义和全局定义的。定义在所有函数外部的结构，则称为全局结构；定义在函数内部的结构，则称为局部结构。和普通变量一样，全局结构和局部结构所被引用的范围不同。

【例 8.18】　局部定义的结构。

```
main ()
{
    struct   /*局部定义*/
    {
```

```
        int a,b;
        char ch;
    }arg;
    arg. a =100;
    fl(arg);
    printf("a =%d\n",arg. a);
}
fl(parm)
struct   /* 形参声明定义 */
{
        int x,y;
        char ch;
}parm;
{       /* 函数体 */
        printf("x1 =%d\n",parm. x);
        parm. x =30;
        printf("x2 =%d\n",parm. x);
        return;
}
```

执行结果:

```
x1 =100
x2 =30
a =100
```

【例 8.19】 全局定义的结构

```
struct st   /* 全局定义 */
{
    int a,b;
    char ch;
};
main()
{
    struct st arg;
    arg. a =1000;
    f1(arg);
}
f1(parm)   /* 定义函数据库 */
struct st parm;   /* 形参声明 */
{
    printf("%d\n",parm. a);
    return;
```

```
}
```

在该程序中，结构类型 st 是全局定义的，在各个函数中都可引用。

8.9.4　结构的初始化

在进行结构声明时，在行尾加上分号，随后在花括号中按结构定义的各成员的顺序给以各自的初始值。例如：

```
struct date
{
    int month;
    int day;
    int year;
};
/*行尾加上分号*/
static struct date today = {11,19,1991};
```

注意，局部结构变量不能进行初始化。

8.9.5　联合

1. 联合（union）的定义声明及其一般格式

不同数据类型的数据可以共用同一个存储区，这是 C 语言中又一种构造类型的数据类型，被称为联合。其定义声明的一般格式如下：

```
union[联合类型名]
{
   类型声明符 变量名;
       ⋮
   类型声明符 变量名;
   }[联合变量列表];
```

例如：

```
union mixed
{
   char c;
   float f;
   short int i;
}x;
```

或者

```
union
{
   char c;
   float f;
   short int i;
}x;
```

或者

```
union mixed
{
    char c;
    float f;
    short int i;
};
union mixed x;
```

从上面可以看出，联合的定义声明与结构很相似，然而，在内存的配合和使用上，两者有本质的区别：上述例子中的联合变量 x 所占字节数应为 3 个成员变量 c、f 和 i 中占用字节数的最大值，即变量 f 所占字节数，可占 4 字节，同一时刻，只能存有 3 个成员变量之一。所以，联合的好处是可以节省空间。

2. 联合的存取

联合成员的引用，在表示方法上，类似于结构成员的引用，其一般形式如下：

联合类型变量名 . 成员变量名

第 9 章　AVR 开发套件快速入门

9.1　AVR 单片机实验系统简介

“AVR-MEGA16 综合学习系统”（见图 9-1）集成常用的单片机外围硬件，提供 ISP 程序下载和 JTAG 仿真接口。系统附带的众多实例程序，可以让用户在最短的时间内，全面地了解掌握 AVR 单片机编程技术。特别适合于单片机初学者、大中专院校实验室选用。

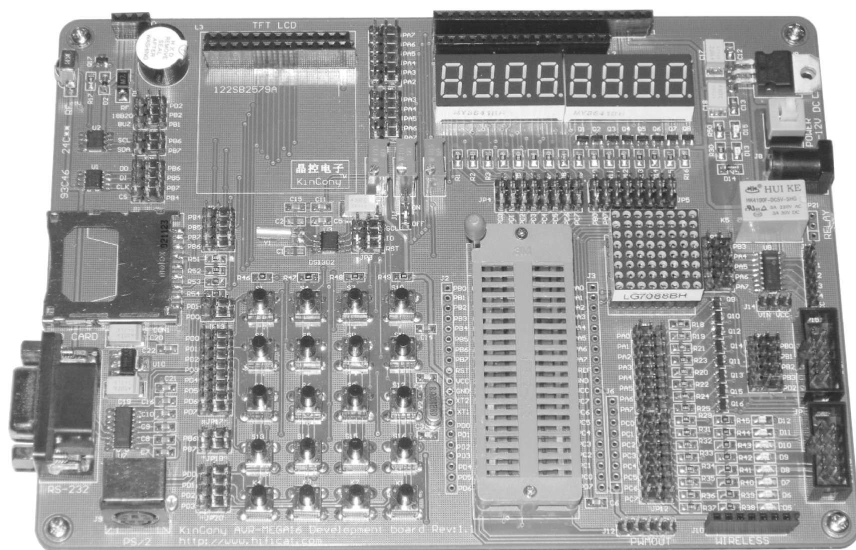


图 9-1　AVR-MEGA16 综合学习系统

其主要特点如下：

1）板载 ISP、JTAG 接口，配上下载线和仿真器，可实现在线烧写和仿真功能。

2）串口通信，支持 USB 转 RS232 串口线，可以直接用于只有 USB 接口的笔记本电脑或台式计算机。

3）集成 6 芯 PS/2 接口，支持个人计算机接口标准 104 键盘等多合一功能，支持光耦合器。

4）试验单元功能采用跳线方式连接，I/O 口使用灵活、方便，使用杜邦线可以自行任意扩展。

5）板载丰富的外扩接线端子：A/D 输入端，继电器常闭、常开端，无线模块接口，PWM 输出接口。

6）丰富的板载资源配备：集成了基本上所有单片机应用中可能遇到的功能模块部分，用户可不必再去找其他零件，即可轻松完成所需要的开发任务。

7）开放性设计，可扩接任意功能的外围模块：单片机的 40 脚接口全部外留，外围模

块还可以不断添加，用户也可以自己动手做外围模块，具有相当的升级潜力。

AVR-MEGA16 综合学习系统包含了常见的经典单片机实验硬件单元，完全兼容教科书及网络上的各类单片机教程，包含的试验单元和接口单元如下：

1）8 位 LED 数码管：可以试验和仿真各种计数器、数字显示，以及用单片机做电子钟等仿真，比如计数器、秒表、电子钟等。

2）8 路 LED：可以显示 PC 口的状态，以及试验和仿真各种 LED 实验，比如正、反向流水灯，交通指示灯等。

3）4 ×4 矩阵键盘：共 16 个键位，可以试验和仿真相关教程的键盘有关的程序。

4）4 个直控键盘：共 4 个键位，非常实用的键盘，通过简洁的程序即可完成键盘输入控制，编程方面更不需要像矩阵键盘那样绞尽脑汁。

5）音乐输出蜂鸣器扬声器：用一个 PNP 型晶体管去控制无源蜂鸣器，用于发出声音、程序报警或播放音乐。

6）继电器试验：有了它就可以知道怎么来做一个以弱控强的系统。以弱控强器件是工控最常用器件之一，与其他驱动器件相比明显的优点是，抗过载能力强，强弱端隔离能力强。

7）I^2C 串行 EEPROM　24C02：用来做 I^2C 通信实验。

8） SPI EEPROM 93C46：用来做 SPI 通信实验。

9）160X 液晶屏：2 行每行 16 个字符。自带字符库、带背光，经典的液晶显示器件通过液晶屏显示想要的信息，比发光管、数码管显示更为漂亮，更专业。

10）像素为 128 ×64 图形液晶接口：可以用来显示中文和图形。

11）红外接收头接口：可以做红外线解码实验、红外线遥控器等。配合遥控器完成遥控解码及红外遥控实验。如按遥控器上的按键，即可点亮实验板上相应的发光管。当然，也可以通过改动程序来达到红外遥控其他资源的目的。

12）所有芯片引脚都接有外扩排针：有利于外扩更多的功能，外扩实验的功能没有限制，完全由用户决定。

13）支持 PS/2 接口的 104 键标准键盘的解码试验：板上的 PS/2 接口是一个足以让常用按钮键盘淘汰的强悍接口。通过随机光盘中的例程，用户学习使用 PS 接口键盘。

14）板上含有步进电动机驱动接口，可以非常方便地接上步进电动机，完成步进电动机的各类实验，如电动机的正、反转等，智能电压调节电路以方便连接不同工作电压的步进电动机。

15）8 ×8 点阵：各点亮度均匀、充足，可显示图形和文字，显示图形或文字稳定、清晰、无串扰，图形或文字显示有静止、移入移出等显示方式。

16）TFT 彩屏接口：板载标准 SD/MMC 卡座、1. 8in[⊖] TFT 真彩液晶屏、3. 3V 稳压器，完成单片机驱动 TFT 真彩液晶实验，制作数码相框等。

17）SD 卡接口：完成 SD 卡的读写实验。

18）PWM 输出接口：完成 PWM 输出实验。

19）串行时钟芯片 DS1302：一种比较常见的 SPI 串行时钟芯片。

20）温度传感器 DS18B20 接口：通过这个接口连好 DS18B20 后，可以实现对温度的高

⊖　1in = 0. 0254m

精确测量，通过多个 DS18B20 传感器也可以做一个多点的温度采集系统，它属于工业环境中常见的一种高精度温度传感器。

21）串口通信电路：使用 MAX232 芯片作串口通信控制，标准 RS232 接口是与 PC 完成联机通信的接口。

22）支持 USB 转 RS232 转接线：完成串口通信，可以直接用于只有 USB 接口的笔记本电脑或台式计算机。

23）ADC：A/D 转换电路，其中接有可调电阻，用于分压及测试。

24）DAC：D/A 转换电路，实现 DDS 功能。可以用于产生正弦波、锯齿波、方波以及其他波形模拟信号等。

AVR-MEGA16 综合学习系统主要硬件接口功能说明：

RS232 串口：完成串口通信实验。

ISP 接口：标准的 10 针 ISP 下载接口，用于 AVR 程序的下载。

JTAG 接口：JTAG 在线仿真调试接口，使用 JTAGICE 进行在线调试。

AVR-MEGA16 单片机开发板模块如图 9-2 所示。

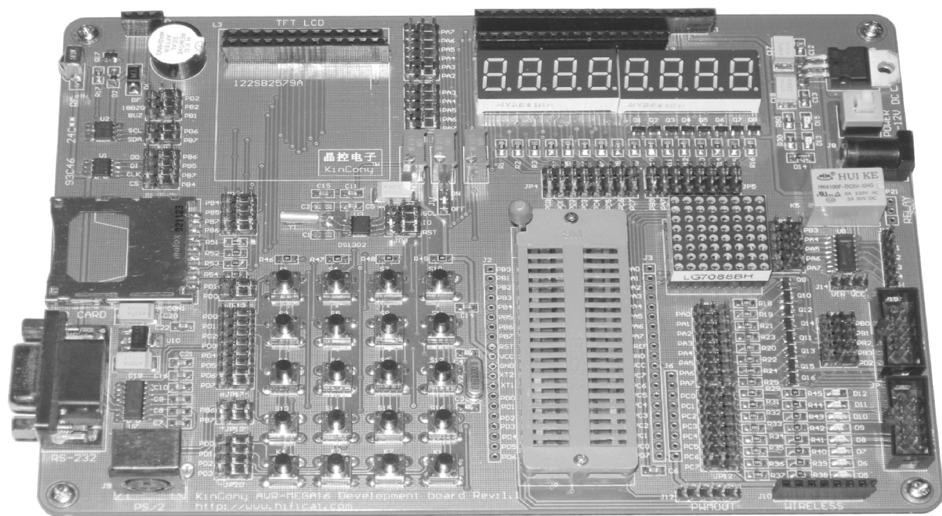


图 9-2　AVR-MEGA16 单片机开发板模块图

各功能模块说明如下：

1）DS18B20 温度传感器接口，直接将 DS18B20 插上即可。

2）蜂鸣器；

3）DS1302（串行时钟芯片）；

4）继电器；

5）93C46（SPI 总线）芯片；

6）24C02（I^2C 总线）芯片；

7）SD、93C46、24C02、跳线引脚，使用对应模块时，用短路帽短接右侧；

8）TFT 彩屏接口，不使用时，最好将彩屏从主机系统上取下保存；

9）DS1302、A/D 转换器、步进电动机、步进电动机电源、无线收发跳线引脚，使用对

应模块时，用短路帽短接右侧；

10）LCD12864、LCD1602 跳线引脚。LCD12864 短路帽短接所有右侧针脚，LCD1602 短路帽短接右侧前 3 排针脚；

11）TFT 跳线引脚，使用 TFT 彩屏时，用短路帽短接右侧；

12）3 个电位器从左到右的功能：调节 A/D 模拟量的采样、调节 LCD12864 液晶亮度、调节 LCD1602 液晶亮度；

13）8 位共阳数码管；

14）LCD1602 字符型液晶屏接口，不使用时，最好将液晶屏从主机系统上取下保存；

15）数码管跳线引脚，使用数码时，将上面两排针脚用短路帽短接；

16）LCD12864 点阵型液晶屏接口，不使用时，最好将液晶屏从主机系统上取下保存；

17）点阵屏；

18）MAX232 串口通信控制芯片；

19）电源开关按键；

20）外接电源接口，额定电压，9 ~ 12V 交流或直流电源均可，容量至少 500mA；

21）步进电动机驱动芯片；

22）步进电动机接口，与步进电动机相连，完成步进电动机正反转及调速实验。连接方法：红线与板上 VCC 对应；

23）ISP 接口，可实现程序在线烧写；

24）点阵屏跳线引脚，使用时将右侧用短路帽短接；

25）JTAG 接口，可实现仿真功能；

26）8 路 LED 灯；

27）无线接收模块接口，完成无线收发功能模块，插时要注意引脚的顺序；

28）PWMOUT 接口，完成 PWM 输出实验；

29）芯镀金锁紧座，可插实验芯片或仿真模块；

30）4 × 4 矩阵键盘；

31）4 位直控键盘；

32）PS/2 标准 PC 键盘接口；

33）RS232 串行接口。A：做仿真操作时使用此接口。B：做单片机与电脑通信的实验时请将随机所配串口线一端插在这里，另一端连接电脑串口；

34）从上到下依次为红外，温度传感器、蜂鸣器跳线引脚，使用对应模块时，用短路帽短接右侧；

35）直控按键、PS2、矩阵按键和串口通信跳线引脚，使用对应模块时，用短路帽短接右侧；

36）SD 卡接口，不使用时，最好将 SD 卡从主机系统上取下保存；

37）红外线遥控接收头，完成红外线解码及红外线遥控等相关实验。

9.2　建立第一个项目（软件操作指南）

打开配套光盘中“ICCV7 for AVR. exe”文件，在“Project（工程）”菜单中执行“NEW（新建）”命令，新建工程文件名为“led. uv2”。接下来在“FILE（文件）”菜单中选择

“NEW（新建）”命令，在弹出的文本编辑框内键入以下代码，即点亮第 1 个 LED 所需要的程序代码，如图 9-3 所示。

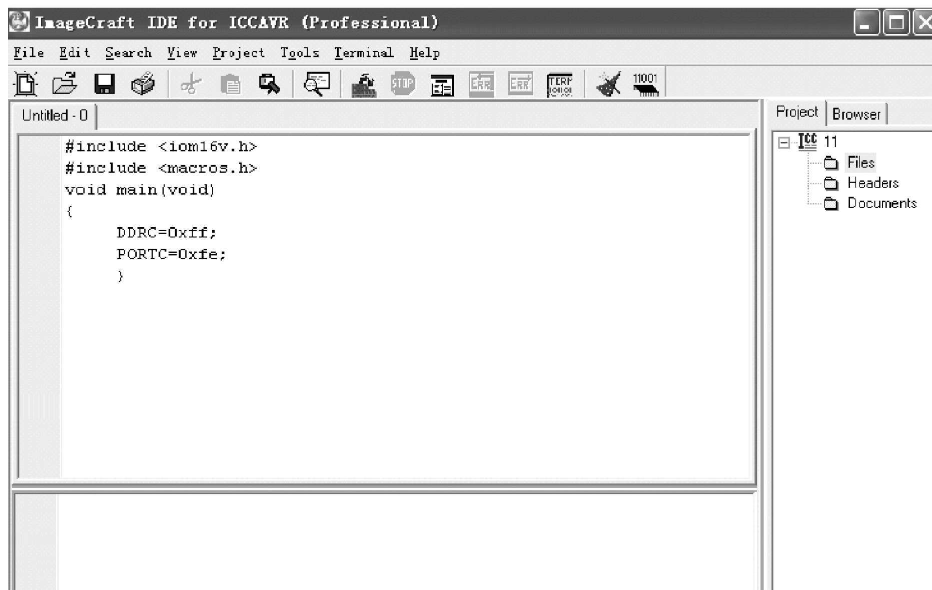


图 9-3　ICCAVR 软件程序编写界面

```
#include <iom16v.h>
#include <macros.h>
void main(void)
{

DDRC = 0xff;
     PORTC = 0xfe;

}
```

这里仅使用了两条语句，DDRC = 0xff 的作用是定义单片机 PC 口的输入输出模式，“1”为输出。PORTC = 0xfe 是使 PC 口的第 8 个引脚为低电平，因为要使一个发光管点亮。从电路图中可以看到，只要使 PC 脚上为低电平信号即可。第 1、2 行是包含头文件语句。一个最简单的程序就这样编写完成了，下面得保存已经编好的程序，即执行“文件”菜单中的“另存为”命令，文件名在此取为 led. c，注意 . c 是 C 语言的扩展名，如果使用汇编语言编写的话，则扩展名应是 . asm。在此，先使用 C 语言来介绍，如图 9-4 所示。

现在已经保存好了这个文件，还记得吗，刚才新建了一个叫做“led”的工程，而 led. c 文件应该是 led 这个工程的其中一份子，换句话说，还应该把这个 led. c 文件添加到 led 这个工程中去。具体操作如下，右键点击屏幕右侧的 Files 字样，则会弹出一个选项，选择“Add Files”选项，把刚才保存好的 led. c 加进去，如图 9-5 所示。

接下来，要为源程序做一项编译工作，即产生目标文件，然后要把该文件烧写到 ATmega16 单片机芯片中去。在执行编译之前，需要进行一些设置，右键点击屏幕右侧的 Files

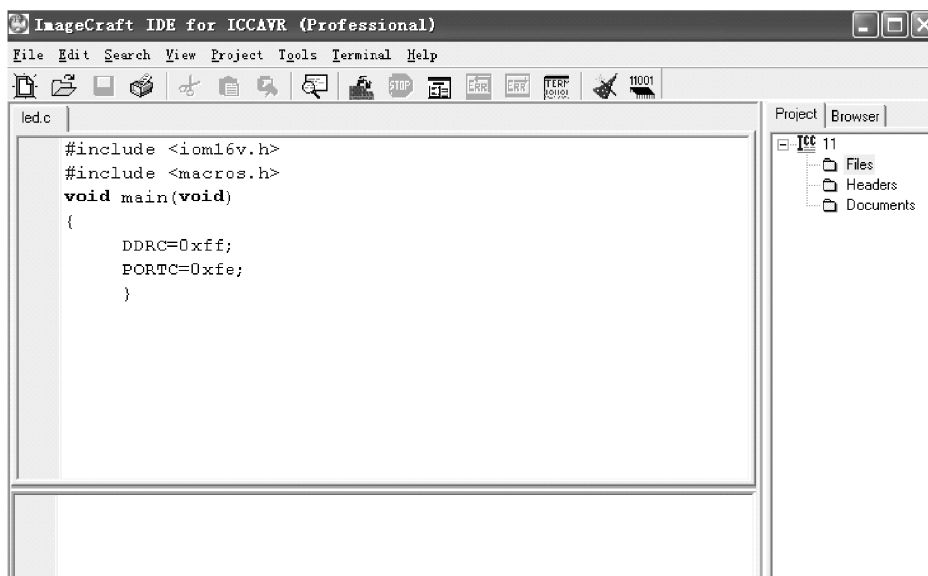


图 9-4　ICCAVR 软件程序编写界面（C 语言）

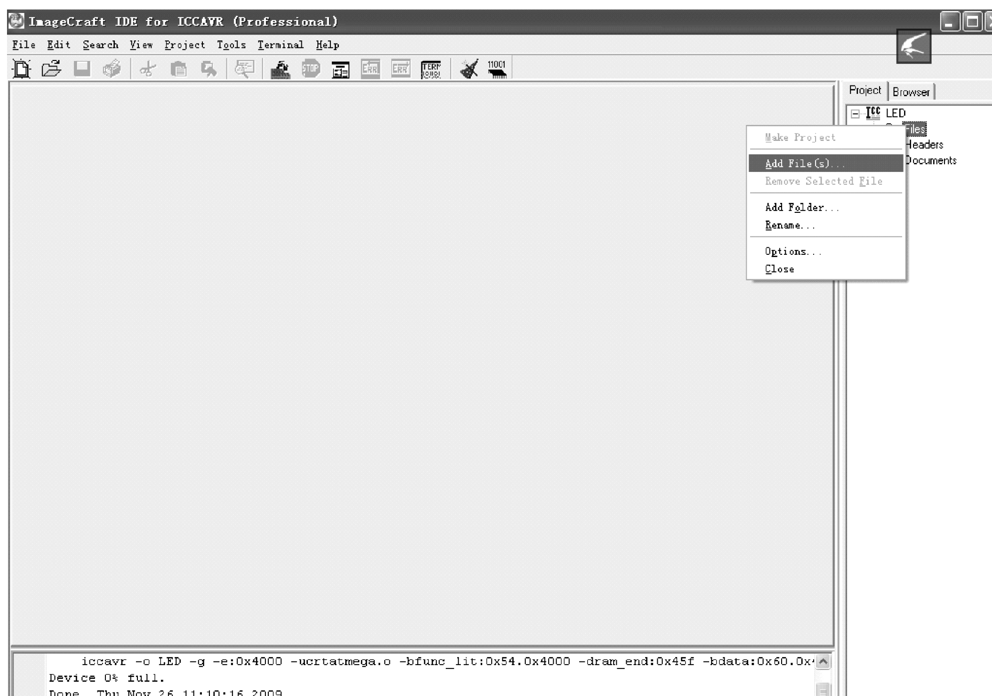


图 9-5　工程中加入文件

字样，则会弹出一个选项，选择“Options”，选择“Target”，如图 9-6 所示进行设置，芯片类型选择“ATmega16”。选择“Compiler”，将“Output Format”设置为“COFF/HEX”，如图 9-7 所示。设置好了，现在只要按一下“Build Project”按钮，就可以完成编译工作了，这时会在 led. c 文件所在目录下发现一个名为“led. hex”的文件，这就是所用来完成烧写芯片工作时使用到的目标程序文件，该文件为十六进制文件。

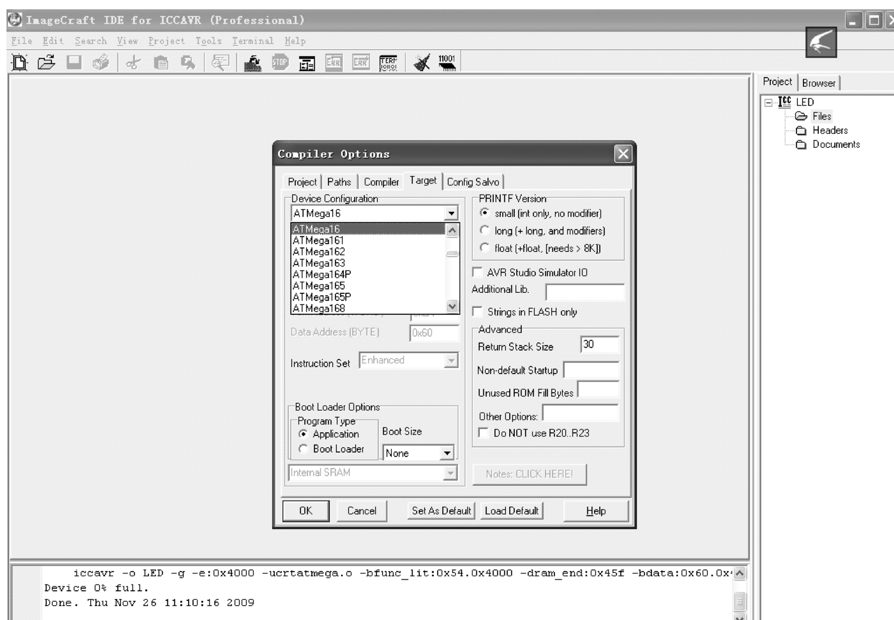


图 9-6　编译选项 1

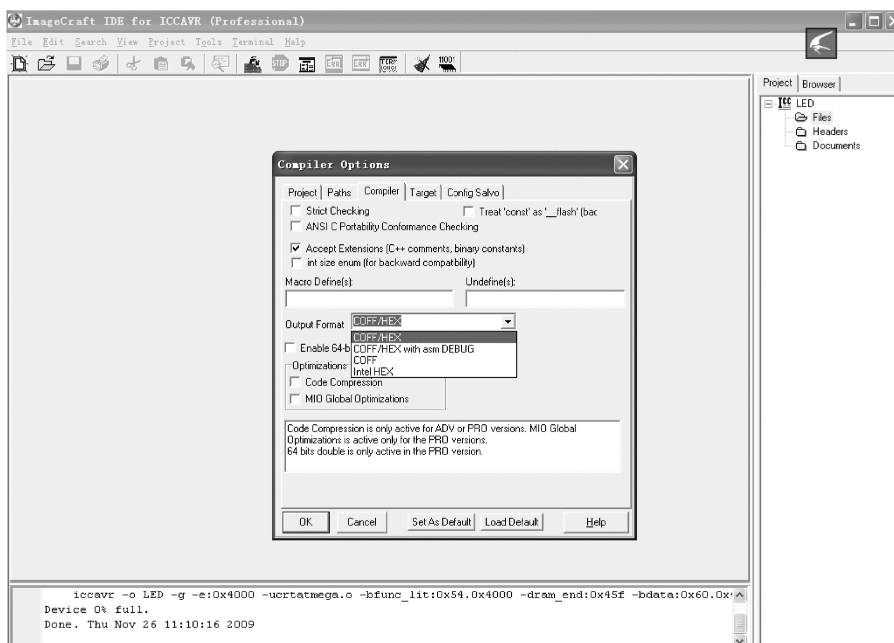


图 9-7　编译选项 2

9.3　AVR 单片机综合学习系统芯片烧写操作指南

现在已经完成了软件程序的编写调试，接下来进行最后一道工序，即程序定形后，如何将其烧写到单片机芯片中去，即如何使用 ISP 在线下载功能。

首先，将 USB ISP 下载线与学习板相连，硬件连接如图 9-8 所示。

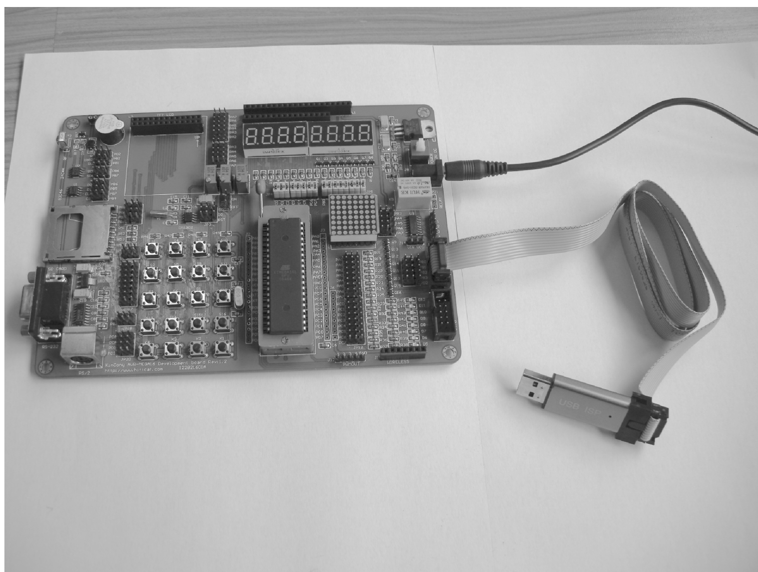


图 9-8　ATmega16 单片机插在 AVR 综合学习系统上

然后打开配套光盘中烧写软件，软件界面如图 9-9 所示。主要操作界面设置如图 9-10 和图 9-11 所示。



图 9-9　并口烧写软件



图 9-10　烧写芯片型号选择

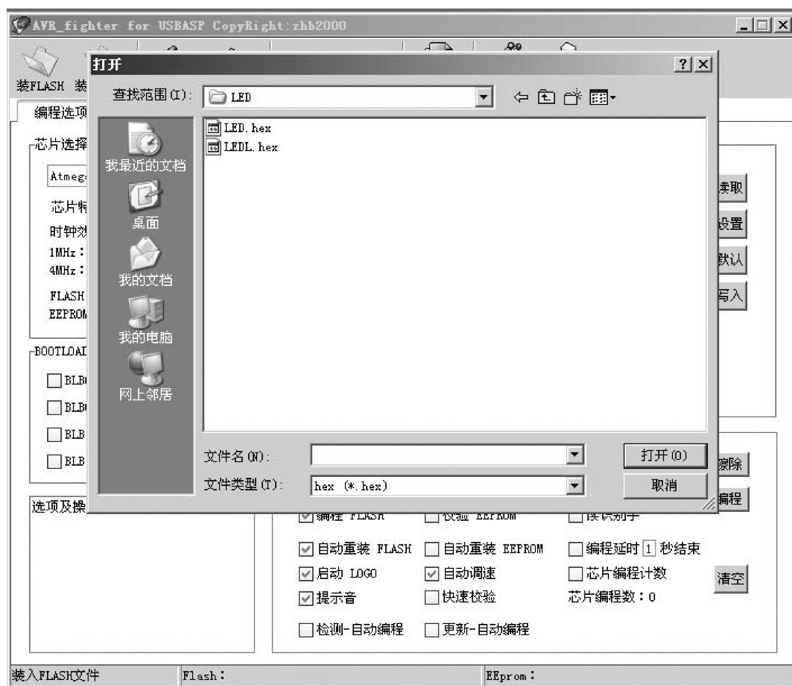


图 9-11　加载烧定文件

下面就需要将要烧写的程序文件调进来，按照前面的软件运行截图，设置好“通信参数设置及器件选择”。Flash 存储器中，将刚才已经准备好的 led. hex 文件选中打开即可。然

后，执行一下“擦除”操作，差不多1s即可完成芯片的擦除工作，速度很快。现在已经到最后一步了，用鼠标点一下“编程”按钮。至此，已经完成了从软件编写，直到烧写芯片的全部步骤。现在就可以看到如图9-12所示的实验验证结果照片了。

图9-12　点亮 LED 效果图

9.4　AVR ATmega16 单片机引脚说明

ATmega16 单片机有3种形式的封装：PDIP（双列直插）、TQFP-44（方形）和 MLF-44（贴片形式）。其外部引脚封装如图9-13所示。

PDIP-40 封装

TQFP/MLF-44 封装

图9-13　封装图

各引脚功能如下。

1. 电源、系统晶振、芯片复位引脚

VCC　　芯片供电（片内数字电路电源）输入引脚，使用时连接到电源正极。

AVCC　　端口 A 和片内 ADC 模拟电路电源输入引脚。不使用 ADC 时，直接连接到电源正极；使用 ADC 时，应通过一个地通电源滤波器与 VCC 连接。

AREF　　使用 ADC 时，可作为外部 ADC 参考电源的输入引脚。

GND　　芯片接地引脚，使用时接地。

XTAL2　　片内反相振荡放大器的输出端。

XTAL1　　片内反相振荡放大器和内部时钟操作电路的输入端。

$\overline{\text{RESET}}$　　芯片复位输入引脚。在该引脚上施加（拉低）一个最小脉冲宽度为 1.5μs 的低电平，将引起芯片的引脚复位。

2. I/O 引脚

I/O 引脚共 32 个，分成 PA、PB、PC 和 PD 等 4 个 8 位端口，它们全部是可编程控制的双（多）功能复用的 I/O 引脚（口）。

端口 A（PA7～PA0）：端口 A 作为 A/D 转换器的模拟输入端，为 8 位双向 I/O 口，具有可编程的内部上拉电阻。其输出缓冲器具有对称的驱动特性，可以输出和吸收大电流。作为输入使用时，若内部上拉电阻使能，端口被外部电路拉低时将输出电流。在复位过程中，即使系统时钟还未起振，端口 A 处于高阻状态。

端口 B（PB7～PB0）：端口 B 为 8 位双向 I/O 口，具有可编程的内部上拉电阻。其输出缓冲器具有对称的驱动特性，可以输出和吸收大电流。作为输入使用时，若内部上拉电阻使能，端口被外部电路拉低时将输出电流。在复位过程中，即使系统时钟还未起振，端口 B 处于高阻状态。端口 B 也可以用做其他不同的特殊功能。

端口 C（PC7～PC0）：端口 C 为 8 位双向 I/O 口，具有可编程的内部上拉电阻。其输出缓冲器具有对称的驱动特性，可以输出和吸收大电流。作为输入使用时，若内部上拉电阻使能，端口被外部电路拉低时将输出电流。在复位过程中，即使系统时钟还未起振，端口 C 处于高阻状态。如果 JTAG 接口使能，即使复位出现引脚 PC5（TDI）、PC3（TMS）与 PC2（TCK）的上拉电阻被激活，端口 C 也可以用做其他不同的特殊功能。

端口 D（PD7～PD0）：端口 D 为 8 位双向 I/O 口，具有可编程的内部上拉电阻。其输出缓冲器具有对称的驱动特性，可以输出和吸收大电流。作为输入使用时，若内部上拉电阻使能，则端口被外部电路拉低时将输出电流。在复位过程中，即使系统时钟还未起振，端口 D 处于高阻状态。端口 D 也可以用做其他不同的特殊功能。

3. ATmega16 单片机的内部结构

图 9-14 是 ATmega16 的结构框图。它是在 AVR 内核的基础上，具体化的一个实例。

从图中可以看出，ATmega16 内部的主要构成部分如下：

1）AVR CPU 部分；

2）程序存储器 Flash；

3）数据存储器 RAM 和 EEPROM；

4）各种功能的外围接口、I/O 口，以及与它们相关的数据、控制、状态寄存器等。

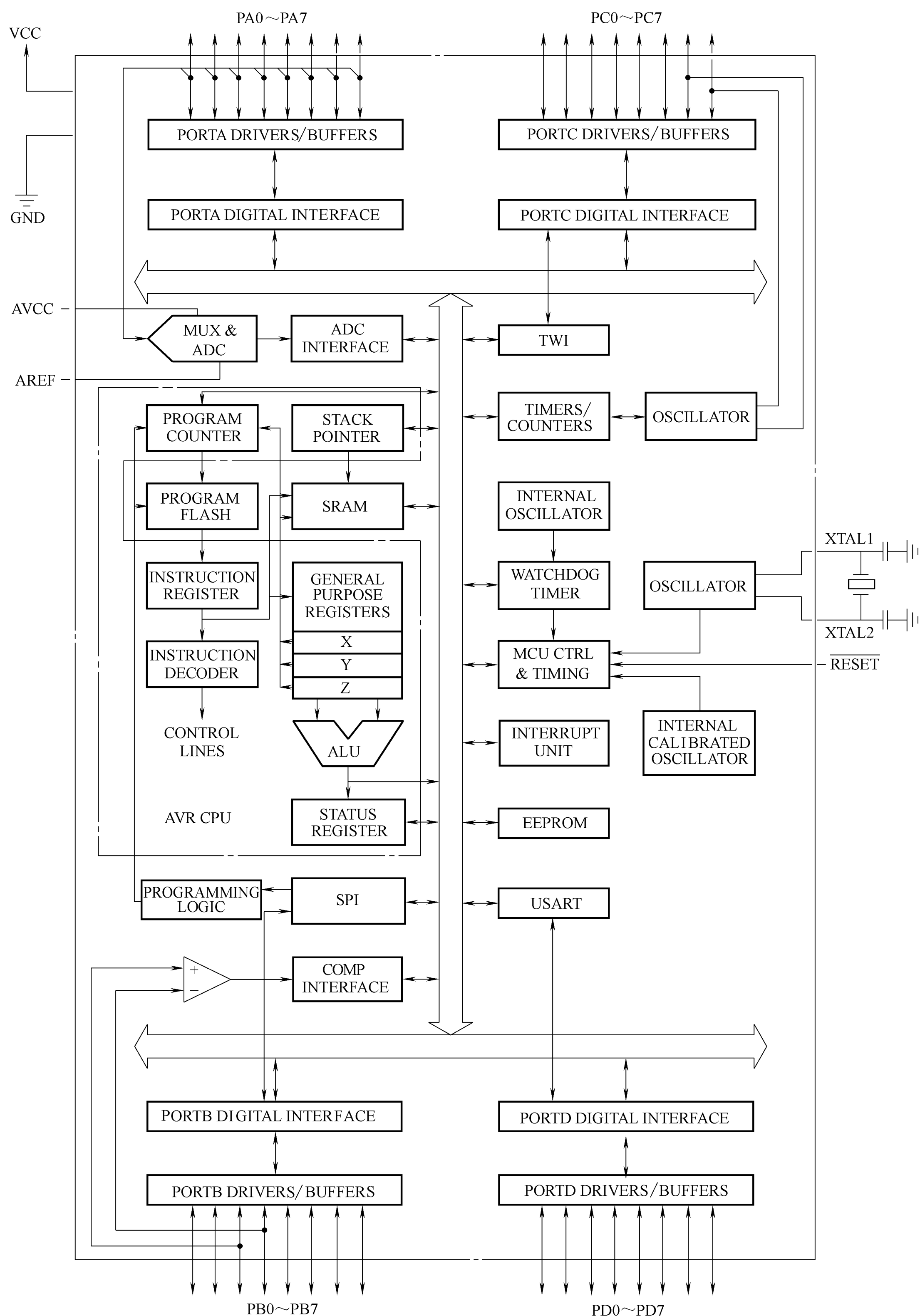


图 9-14　ATmega16 结构框图

第 10 章　ATmega16 基础实例

经过前面的理论学习之后，从本章开始要进入真正的实践过程。结合作者本身的经历，在进入单片机领域之前一直感觉单片机是个非常神奇的东西，写好程序后居然可以看到整个电路能根据自己的思想在运行。虽然在入门之前已经学了基本的软、硬件理论，但想要真正把整个过程顺利执行还是感到一时间无从下手。本章就从最基本的实例讲起，希望读者可以一步一步地跟着实践下去，学完本章也就对 AVR 单片机入门了。

在正式进入基础实例之前，先简单介绍一下开发单片机系统的流程和必备条件。开发单片机的流程主要分为这几步：设计硬件原理图、画软件流程图、编程调试、修改完善。当然要成功开发一个单片机系统首先要有相关的硬件设备，如计算机、编程器等开发工具，其次还要有相关软件的配合，如 ICCAVR、PROTEL 等。对于初学者来说，在学习本章内容的同时要不断地回顾前几章的内容，因为本章所有的实例全部要以前面几章为基础，只有学好了前面几章内容再来学本章才会事半功倍。

10.1　发光二极管闪动实验

发光二极管（简称 LED），在日常生活中经常看到有些电器上带有 LED 指示灯有节奏地闪动，通过这个 LED 指示灯可以了解系统的工作状态。本节就介绍最简单的发光管应用实验。

10.1.1　实例功能

本节介绍发光二极管的器件原理和与单片机之间的应用。让读者了解如何通过单片机的 I/O 口来控制发光管的亮灭。本实例以 PA 口控制 8 个发光二极管为例，说明其应用方法。通过这个基础实例可以掌握 AVR 单片机 I/O 口的应用。

10.1.2　器件与原理

本实验中主要应用到单片机的端口操作及延时循环程序。首先需要知道如何让一个发光二极管工作。发光二极管有很多种类，图 10-1 所示的是几种直径 3mm 的普通亮度发光二极管，电气图形符号如图 10-2 所示，当在它的 A 和 K 两个电极加上合适的电压时，它就会亮



图 10-1　发光二极管实物图

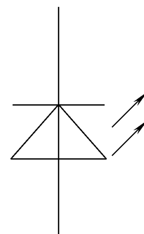


图 10-2　发光二极管电气图形符号

起来。说“合适的电压”，是因为不同的发光二极管工作电压并不相同，一般是在 1.6 ~ 2.8V 之间，而工作电流则一般在 2 ~ 30mA 之间，但是实际工作的选择范围一般是 4 ~ 10mA 之间。

10.1.3　硬件电路

上文介绍的发光管电压、电流参数，实际是为了解释 LED 上串接电阻大小的选择。例如系统供电为 5V，LED 上串接的电阻是 1kΩ，如果此时 LED 上的电压是 2.0V，那么通过 LED 的电流则为（5V－2V）/1000Ω＝3mA，如果需要提高亮度，一般会将电流控制在 10mA 左右，则此时电阻应该选择（5V－2V）/10mA＝300Ω，所以串联电阻可以选择 300Ω。

电阻已经确定，然后就是连接到单片机的 I/O 口上，如图 10-3 所示，可以看到 LED 的 A 极通过限流电阻连接到 VCC，K 极连接到了单片机的 I/O 口，因此要使 LED 发光，也就是使电流流过 LED，只需要把 I/O 口置成低电平即可，所以最终对 LED 的控制变成了对一个 I/O 口的控制，比如要点亮 LED，就是把 PA 口设置成低电平而已，这就是实现方法。实验电路可以参考图 10-3。

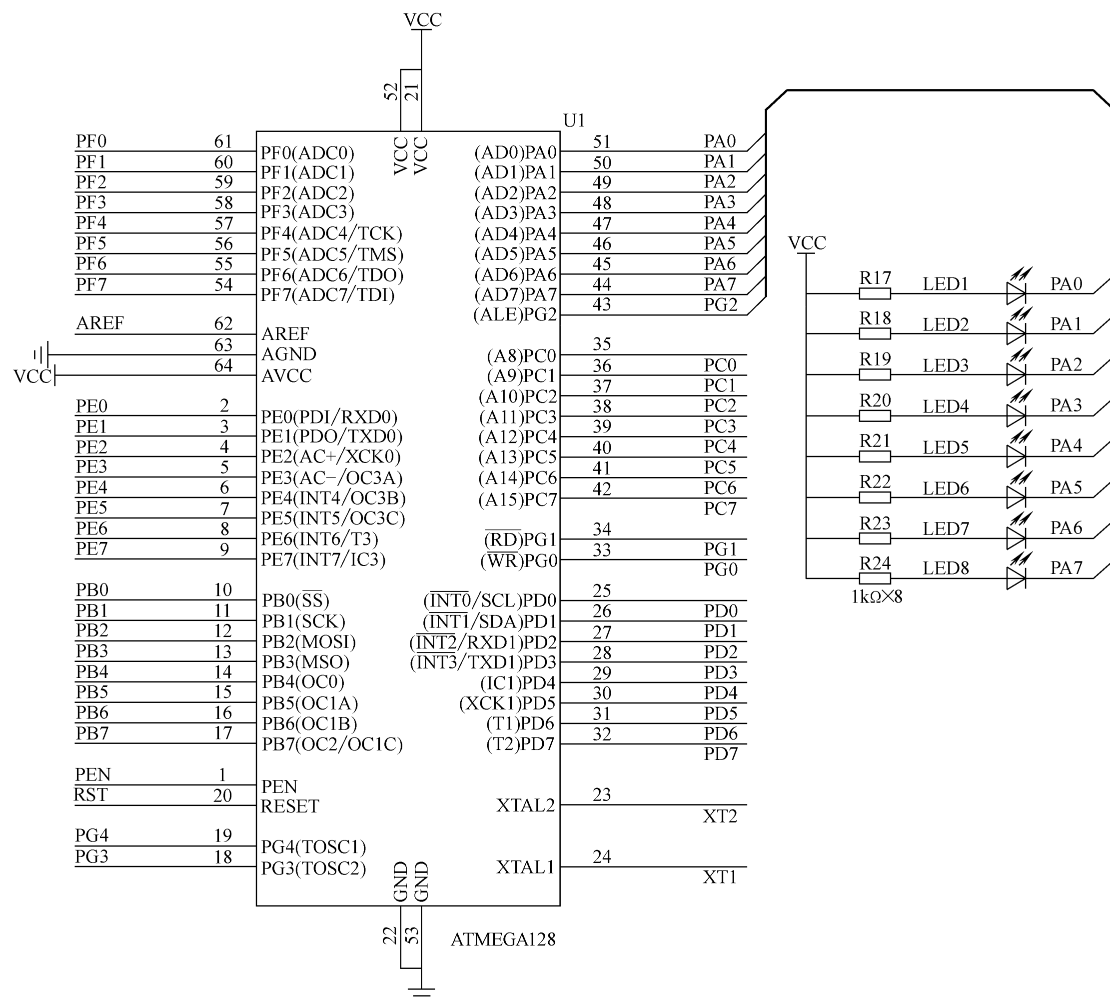


图 10-3　发光二极管应用电路

10.1.4　程序设计

通过上文已经知道，要使图 10-3 中的 LED 发光，只要把 PA 口置成低电平就可以了，相反，把 PA 口置成高电平就可以使 LED 灭掉，而要想让它闪动起来，其实也就是亮和灭在一段连续时间上交替出现。所以，实现方法就是使 PA 口在每隔一段时间轮流出现高、低电平。因为单片机的程序执行速度很快，如果是在很短的时间内改变 PA 口的状态，人眼是看不出来的，中间必须有个合适的延迟时间。到这里就可以编写如下程序，通过以下程序可以实现 LED 的闪动发光。

程序代码如下：

```
/*************************************************************************/
/* LED 闪动测试程序                                                       */
/* 目标器件:ATmega16                                                      */
/* 晶振:RC 1MHz                                                           */
/* 编译环境:ICCAVR 6.31A                                                  */
/*************************************************************************/

/************************** 包含头文件 **************************/
#include <iom16v.h>
#include <macros.h>

/*********************************************************************
函数功能:延时子程序
入口参数:
出口参数:
**********************************************************************/
void delay(void)
{
    unsigned int k;
    for(k=0;k<5000;k++);
}

/*********************************************************************
函数功能:主程序
入口参数:
出口参数:
**********************************************************************/
void main(void)
{
    DDRA=0xff;
    PORTA=0xff;
```

```
    while(1)
    {
      PORTA^ =0x01;       //PA0 不断延时变化
      delay( );
      delay( );
      delay( );
    }
}
```

10.2　流水灯实验

看了以上代码后很多初学者都觉得很好理解，LED 发光管控制也是很多单片机入门的第一个实例，程序虽然简单但对初学者具有很大意义。很多教材上也提到了流水灯的应用，其实它与我们上面程序大同小异，它的原理是使每个 I/O 口轮流输出低电平，那么相应端口上的发光管轮流发光，在 PCB 上把这几个 LED 按顺序排列的话就可以看到流水灯的效果了。

在使用配套开发板来实现流水灯的原理是：使 8 个编号为 LED1 ~ LED8 的 LED 从 LED1 开始亮起，每次只点亮一个，并按次序往 LED8 移动，结束后再次从头开始。参照表 10-1，实际上就是在程序开始执行之后，使程序一直在“复位状态”到“状态 8”之间按顺序执行。如果把这些文字整理成一个状态表的话更有助于初学者理解，流水灯的状态变化见表 10-1。

表 10-1　流水灯状态表

LED 序号	LED1	LED2	LED3	LED4	LED5	LED6	LED7	LED8
对应的 I/O 口	PA0	PA1	PA2	PA3	PA4	PA5	PA6	PA7
复位状态								
状态 1								
状态 2								
状态 3								
状态 4								
状态 5								
状态 6								
状态 7								
状态 8								

从前文中已经知道了控制一个 LED 的方法，所以要实现这个程序，关键是了解整个过程，先来比较一下“复位状态”和“状态 1”的区别，有什么地方不一样呢？就是 LED1 在这里被点亮了，所以从“复位状态”到“状态 1”，需要完成的操作是“点亮 LED1”，然后从“状态 1”到“状态 2”，很容易发现，区别有两个地方，就是 LED2 亮了，LED1 却灭

了，所以在这一步，需要做的事情是“点亮 LED2”和“熄灭 LED1”，按照这样的过程其他步骤需要完成的任务也不难看出来，列出流程图如图 10-4 所示。

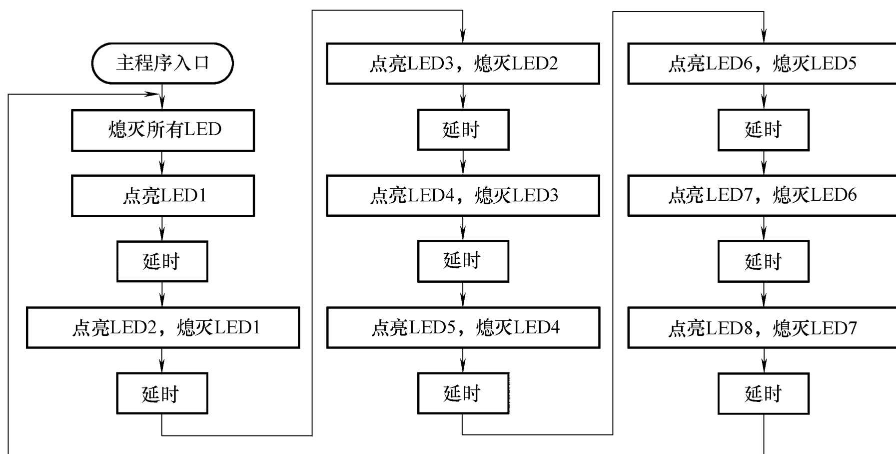


图 10-4　流水灯程序流程图

对应流水灯的流程图，则可以得到如下程序代码：

```
/ ************************************************************************** /
/ * 流水灯测试程序                                                          * /
/ * 目标器件:ATmega16                                                       * /
/ * 晶振:RC 1MHz                                                            * /
/ * 编译环境:ICCAVR 6.31A                                                   * /
/ ************************************************************************** /

/ ************************** 包含头文件 ************************** /
#include <iom16v.h>
#include <macros.h>

/ ************************************************************************
函数功能:延时子程序
入口参数:
出口参数:
************************************************************************ /
void delay(void)
{
   int i;
   for(i = 0;i < 20000;i++);
}

/ ************************************************************************
函数功能:端口初始化程序
```

```
入口参数:
出口参数:
 ************************************************************************/
void port_init (void)
{
   DDRA = 0XFF;
   PORTA = 0XFF;
}

/ ************************************************************************
函数功能:主程序
入口参数:
出口参数:
 ************************************************************************/
void main (void)
{
   port_init();
   while(1)
   {
     PORTA = 0xFE;      //PA0 亮
     delay();
     PORTA = 0xFD;      //PA1 亮
     delay();
     PORTA = 0xFB;      //PA2 亮
     delay();
     PORTA = 0xF7;      //PA3 亮
     delay();
     PORTA = 0xEF;      //PA4 亮
     delay();
     PORTA = 0xDF;      //PA5 亮
     delay();
     PORTA = 0xBF;      //PA6 亮
     delay();
     PORTA = 0x7F;      //PA7 亮
     delay();
   }
}
```

相信到这里，读者已经看到发光二极管在流动发光了，单单是对这个程序要求的功能来讲，其实也已经没有什么可以挑剔的，而且通过对这个程序实现的分析，相信凭读者自己的能力，也一定可以写出另外花样的流水灯了，但是也许读者也看到过别人用更加简单的代码

来实现这个流水灯的功能，那就意味着，这个程序还可以按照另外的思路来做。比如如下的程序代码：

```
/*****************************************************************************/
/* 流水灯测试程序                                                              */
/* 目标器件:ATmega16                                                          */
/* 晶振:RC 1MHz                                                               */
/* 编译环境:ICCAVR 6.31A                                                      */
/*****************************************************************************/

/*************************** 包含头文件 ***************************/
#include <iom16v.h>
#include <macros.h>

/*************************************************************************
函数功能:延时子程序
入口参数:
出口参数:
*************************************************************************/
void delay(void)
{
   int i;
       for(i=0;i<20000;i++);
}

/*************************************************************************
函数功能:端口初始化程序
入口参数:
出口参数:
*************************************************************************/
void port_init (void)
{
   DDRA=0XFF;
   PORTA=0XFF;
}

/*************************************************************************
函数功能:主程序
入口参数:
出口参数:
*************************************************************************/
```

```
void main (void)
{
    unsigned char temp;
    unsigned char i;
    DDRE =0x00;
    DDRG =0xff;
    port_init( );
    PORTA =0xff;               //关所有 LED
    while(1)
    {
    for(i =0;i <8;i + + )
       {
          PORTA =0xff;
          temp =1 < <i;
          PORTA =PORTA&( ~temp);      //PA 口轮流输出
          delay( );
       }
    }
}
```

这个程序实现的功能与上文程序实现的功能完全一致，其最大的不同是在思路上，它已经不再单独地来看某个 LED 了，所以程序中也不再定义 LED 的连接，可以认为它将 8 个 LED 合成一个整体来考虑，也可以说，其实它就只关心 PA 口，因为 LED1 ~ LED8 本来就是与 PA0 ~ PA7 对应的，所以程序的目的是在 PA 口上实现这样的功能：PA0 ~ PA7 按顺序循环使某个口线为低电平，同时其他口线为高电平，再进一步，现在 PA 口的数据和变量 temp 是对应起来的，所以最终对 8 个 LED 的操作变成了对变量 temp 操作。用流程来看一下 temp 的变化，也就明白整个过程了，见表 10-2。

表 10-2　temp 的变化过程

	i =0	i =1	i =2	i =3	i =4	i =5	i =6	i =7
temp	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80
~ temp	0xfe	0xfd	0xfb	0xf7	0xef	0xdf	0xbf	0x7f

10. 3　按键实验

按键是单片机系统中常用的信息输入部件，同时也是人机对话中不可缺少的输入设备，其外形如图 10-6 所示。在和单片机构成系统时，按键通常有两种接法：一种叫做独立式按键；另外一种叫做行列式或者是扫描式按键。通过本节的学习，可以了解按键电路的应用及程序编写。

10. 3. 1　实例功能

本实例利用独立按键来演示键盘电路的工作原理。用一个按键来控制一个 LED 灯的亮灭。当把 S1 按键按下时 LED 全亮，当把 S1 释放时 LED 全灭。

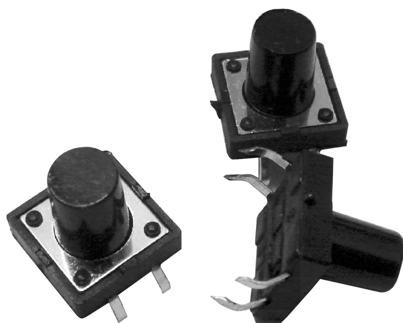


图 10-5　按钮实物图

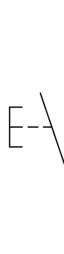


图 10-6　按钮电气图形符号

10. 3. 2　器件与原理

首先来了解一下按键的结构。一般的按键从实物来看，是四端口器件，但是其实它是二端口器件，参照图 10-5 和图 10-6 中的按钮实物就不难明白，在按下按钮之前，两个触点之间是不导通的，按下时就导通，通过外部电路的不同接法，就可以使其中一个端口在按下和不按下时产生电平变化，而单片机正是通过检测到这种变化来完成对按键输入信息的获得。单片机的大部分端口都有内部上拉电阻或 MOS 管，一般都可以实现上拉，所以在按键按下之前，S1 对应的端口 PE4 保持在高电平状态，当按键按下时，PE4 通过 S1 接到 VSS，这时就是低电平。所以，要想在程序里检测到是否有按键按下，关键就是检查对应端口的状态变化，这个就是单片机系统中的按键编程的原理。所以针对程序设计目的，我们的方法就是不断地检测 PE4 的电平状态，然后根据检测结果控制 LED。按键实验原理如图 10-7 所示。

程序代码如下：

```
/****************************************************************************/
/*独立键盘测试程序                                                          */
/*目标器件:ATmega16                                                         */
/*晶振:RC 1MHz                                                              */
/*编译环境:ICCAVR 6.31A                                                     */
/****************************************************************************/

/**************************包含头文件**************************/
#include <iom16v.h>
#include <macros.h>
/*************************************************************************
函数功能:主程序
入口参数:
出口参数:
*************************************************************************/
void main(void)
{
    DDRD = 0x00;//PD 口输入
```

```
    PORTD = 0xff;
    DDRA = 0xff;
    PORTA = 0xff;
    while(1)
    {
    PORTA = PIND;
  }
}
```

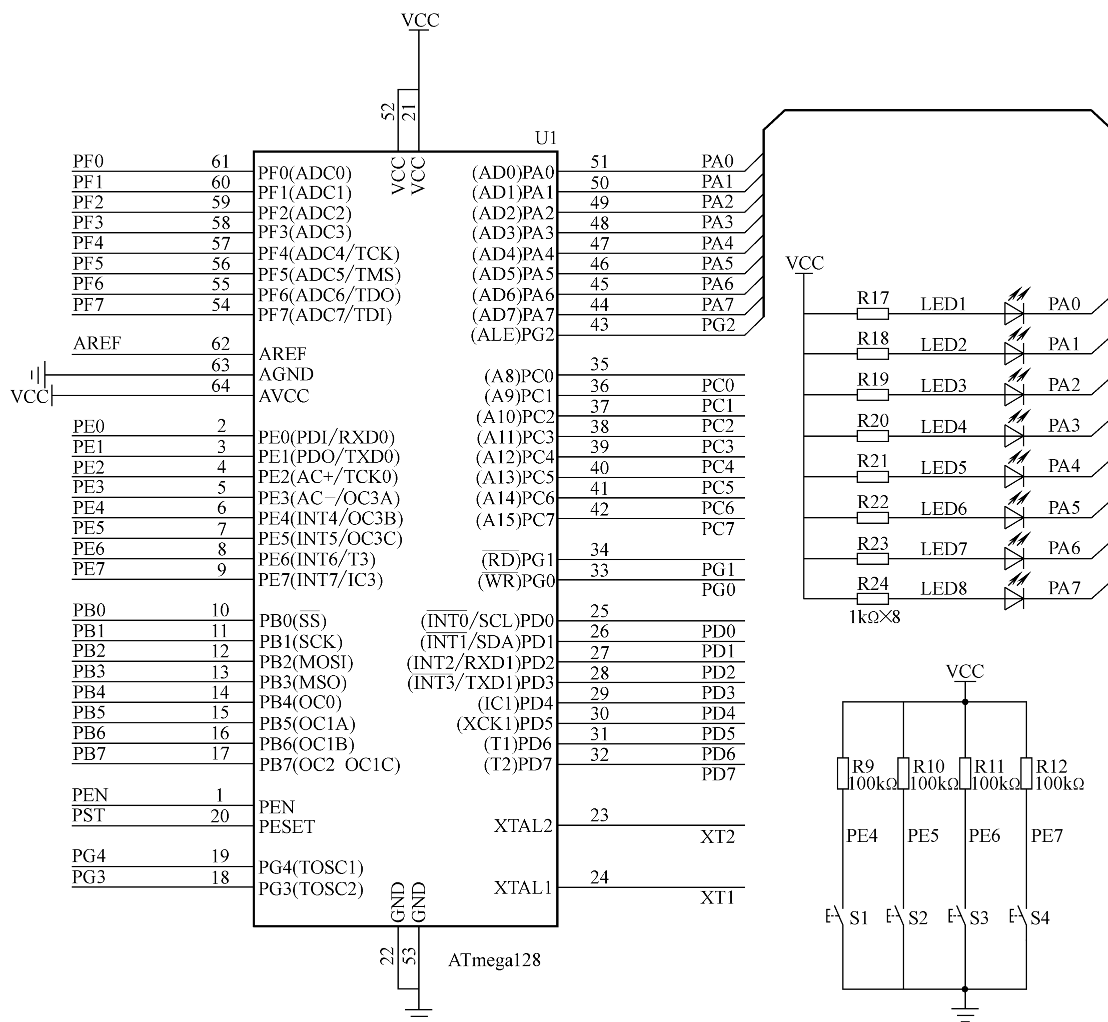


图 10-7　按键实验原理图

10.3.3　程序设计

在上面程序中实现这么简单的功能不用单片机也能轻易做到。但在实际应用中一般都不止一个独立按键，这就要用到按键识别功能了，下面介绍一下独立按键识别的程序，程序中把按下的键值通过数码管显示出来。数码管显示部分内容将在下文介绍。

```
/ ************************************************************************ /
/ * 独立键盘测试程序                                                       * /
/ * 目标器件:ATmega16                                                      * /
/ * 晶振:RC 1MHz                                                           * /
/ * 编译环境:ICCAVR 6.31A                                                  * /
/ ************************************************************************ /

/ ************************** 包含头文件 ************************** /
#include  <iom16v.h>
#include  <macros.h>

/ ********************** 数码管段码表 ************************** /
extern const unsigned char
tab[ ] = { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,
            /*    0    1    2    3    4    5    6    7    8    9   */
                 0x88,0x83,0xC6,0xA1,0x86,0x8E};
            /*    a    b    c  d    e    f   */


/ ***********************************************************************
函数功能:延时程序
入口参数:
出口参数:
*********************************************************************** /
void delay_1us(void)                    //1μs 延时函数
{
asm("nop");
}

void delay_nus(unsigned int n)          //n μs 延时函数
{
unsigned int i=0;
for (i=0;i<n;i++)
delay_1us();
}

void delay_1ms(void)                    //1ms 延时函数
{
unsigned int i;
for (i=0;i<140;i++);
```

```
}

void delay_nms(unsigned int n)          //n ms 延时函数
{
unsigned int i=0;
for (i=0;i<n;i++)
delay_1ms();
}

/********************************************************************
函数功能:显示子程序
入口参数:k
出口参数:
********************************************************************/
void display(unsigned char k)
{
    PORTC=tab[k];
    PORTA=0xef;
    delay_nms(2);
}

/********************************************************************
函数功能:主程序
入口参数:
出口参数:
********************************************************************/
void main(void)
{
    int keyvalue=0;
    DDRD=0x00;//PD 口输入
    PORTD=0xff;
    DDRC=0xff;
    PORTC=0xff;
    DDRA=0xff;
    PORTA=0xff;
    while(1)
    {
      if((PIND&0x0F)!=0xF0)
      {
         delay_nms(10);
```

```
        if((PIND&0x0F)! =0xF0)
        {
          if((PIND&0x01)= =0)
          keyvalue=1;
          if((PIND&0x02)= =0)
          keyvalue=2;
          if((PIND&0x04)= =0)
          keyvalue=3;
          if((PIND&0x08)= =0)
          keyvalue=4;
          NOP();
        }
      }
      display(keyvalue);
    }
  }
```

10.4　蜂鸣器实验

在很多的单片机系统中除了显示器件外经常还有发声器件，最常见的发声器件是蜂鸣器。蜂鸣器一般用于一些要求不高的声音报警及按键操作提示音等场合。蜂鸣器的形状一般如图 10-8 所示。虽然它有自己的固有频率，但是它也可以被加以不同频率的方波，从而编制一些简单的音乐。

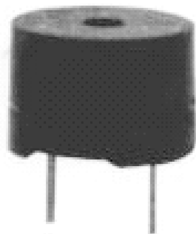


图 10-8　蜂鸣器实物图

10.4.1　实例功能

本实例就是来实现蜂鸣器发声，通过本节的实验，可以使读者熟练掌握蜂鸣器的应用。

10.4.2　器件与原理

蜂鸣器和普通扬声器相比，最重要的一个特点是只要按照极性要求加上合适的直流电压，就可以发出固有频率的声音，因此使用起来比扬声器简单。由此可知，蜂鸣器的控制和 LED 的控制对单片机而言是没有区别的。

10.4.3　硬件电路

虽然蜂鸣器的控制和 LED 的控制对于单片机是一样的，但在外围硬件电路上却有所不同，因为蜂鸣器是一个感性负载，一般不建议用单片机 I/O 口直接对它进行操作，所以最好加一个驱动晶体管，在要求较高的场合还会加上反相保护二极管。本例实验只为了达到学习目的并没有加反相二极管保护，硬件电路可以参考图 10-9 所示。

通过硬件原理图可知，图中晶体管用了 PNP 型，所以要使蜂鸣器发声只要给单片机 PB1 脚置低电平即可，由此可以为下文的程序编写提供关键参考。

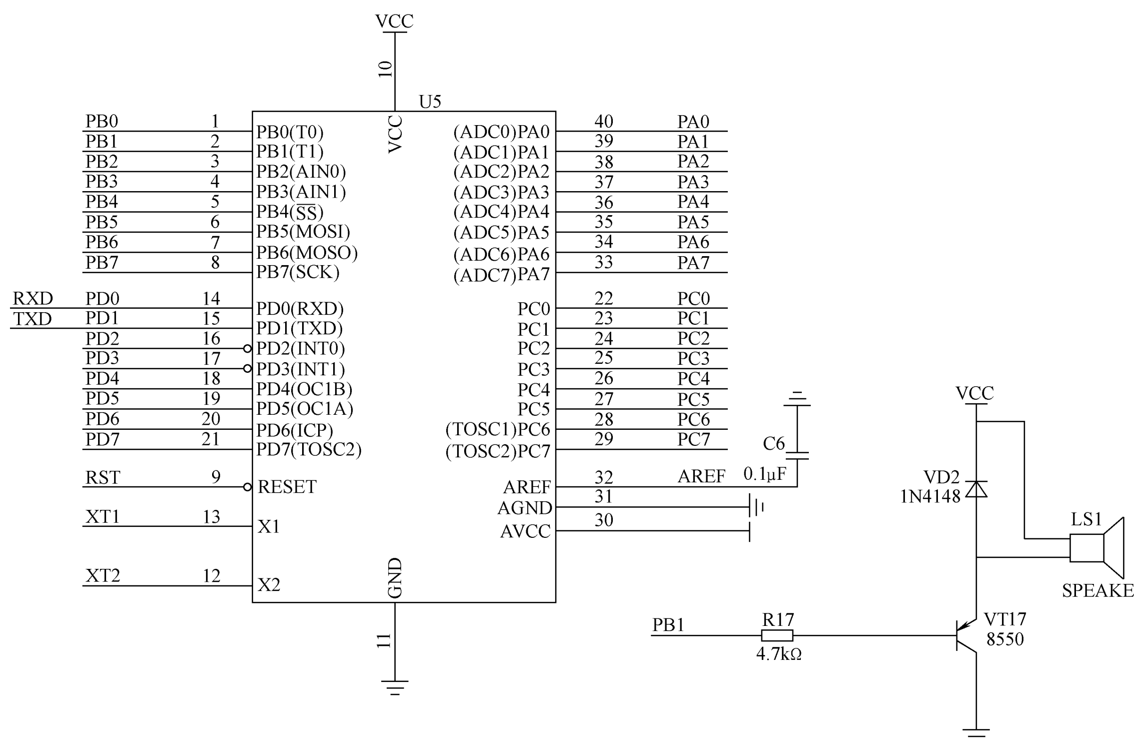


图 10-9　硬件原理图

10.4.4　程序设计

```
/ ************************************************************************ /
/ * 蜂鸣器测试程序                                                    * /
/ * 目标器件:ATmega16                                                 * /
/ * 晶振:RC 1MHz                                                      * /
/ * 编译环境:ICCAVR 6.31A                                              * /
/ ************************************************************************ /

/ *************************** 包含头文件 *************************** /
#include  <iom16v.h>
#include  <macros.h>


/ ****************************** 端口定义 ****************************** /
#define     BellOn()             PORTB &= ~(1 << 1)
#define     BellOff()            PORTB |= (1 << 1)

/ **********************************************************************
函数功能:延时程序
```

```
入口参数:
出口参数:
********************************************************************/
void delay_1us(void)                    //1μs 延时函数
{
  asm("nop");
}

void delay_nus(unsigned int n)          //n μs 延时函数
{
  unsigned int i=0;
  for (i=0;i<n;i++)
  delay_1us();
}

void delay_1ms(void)                    //1ms 延时函数
{
  unsigned int i;
  for (i=0;i<140;i++);
}

void delay_nms(unsigned int n)          //n ms 延时函数
{
  unsigned int i=0;
  for (i=0;i<n;i++)
  delay_1ms();
}

/********************************************************************
函数功能:主程序
入口参数:
出口参数:
********************************************************************/
void main(void)
{
  DDRB=0xff;
  PORTB=0xff;

  while(1)
  {
```

```
        BellOn();
        delay_nms(500);
        BellOff();
        delay_nms(500);
    }
}
```

10.5　继电器实验

在现代自动控制设备中，都存在一个电子电路（弱电）与电气电路（强电）的互相连接问题，一方面要使电子电路的控制信号能够控制电气电路的执行元件（如电动机、电磁铁、电灯等），另一方面又要为电子电路与电气电路提供良好的电隔离，以保护电子电路和人身的安全。继电器便能完成这一桥梁作用。

10.5.1　实例功能

本实例通过单片机来控制继电器吸合、释放，读者可以熟练掌握继电器的使用方法。在本例中读者也可以用继电器的常开、常闭触点控制电灯的亮灭，实现“以小控大”。

10.5.2　器件与原理

继电器是一种电子控制器件，它具有控制系统（又称输入回路）和被控制系统（又称输出回路），通常应用于自动控制电路中，它实际上是用较小的电流去控制较大电流的一种“自动开关”。故在电路中起着自动调节、安全保护、转换电路等作用。在大多数的情况下，继电器就是一个电磁铁，这个电磁铁的衔铁可以闭合或断开一个或数个触点。当电磁铁的绕组中有电流通过时，衔铁被电磁铁吸引，因而就改变了触点的状态。继电器一般可以分为电磁式继电器、热敏干簧继电器、固态继电器等。本实验板上配置的继电器如图 10-10 所示。

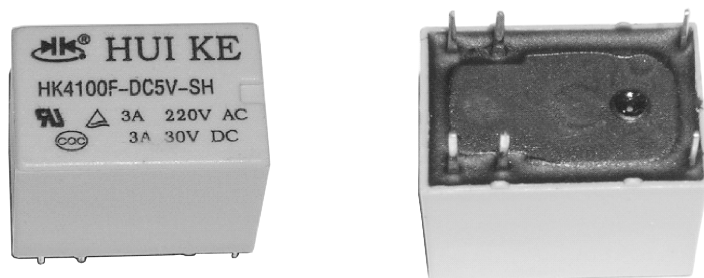


图 10-10　继电器实物图

继电器也是属于感性器件，所以不能用单片机的 I/O 口直接来控制，且要在晶体管等控制器件上加反相保护电路。一般实验中都是单片机通过一个 PNP 型晶体管，把晶体管作为电子开关来驱动继电器，继电器的开和关完全由晶体管的基极电平进行控制。当晶体管基极为高电平，PNP 型晶体管截止，这时继电器不工作；反之为低电平的话，PNP 型晶体管导通，继电器得电吸合。

10.5.3　硬件电路

继电器实验原理图可以参考图 10-11 所示。

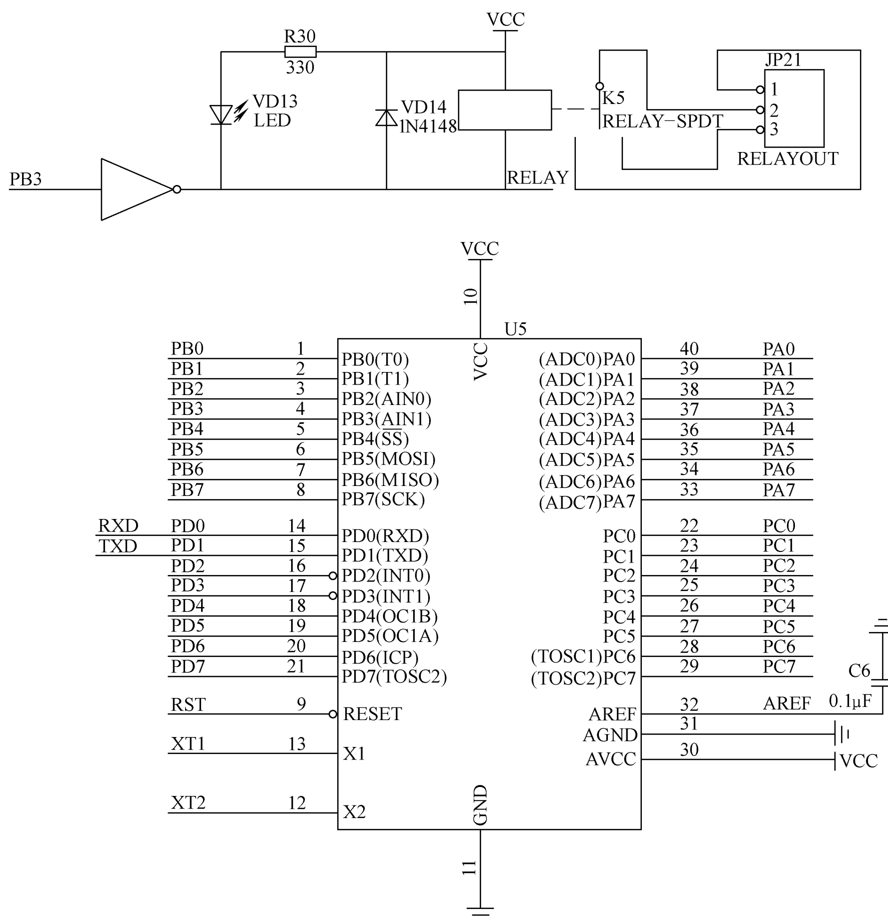


图 10-11　硬件原理图

10.5.4　程序设计

```
/ ***************************************************************************** /
/ * 继电器测试程序                                                            * /
/ * 目标器件：ATmega16                                                        * /
/ * 晶振：RC 1MHz                                                             * /
/ * 编译环境：ICCAVR 6.31A                                                    * /
/ ***************************************************************************** /

/ ************************** 包含头文件 ************************** /
#include <iom16v.h>
#include <macros.h>
```

```
/ ****************************** 端口定义 ********************************* /
#define      RRLAY_On( )            PORTB & = ~(1 < <3)
#define      RELAY_Off( )           PORTB | =(1 < <3)

/ **********************************************************************
函数功能:延时程序
入口参数:
出口参数:
 ********************************************************************** /
void delay_1us( void)                    //1μs 延时函数
{
  asm( " nop" ) ;
}

void delay_nus( unsigned int n)       //n μs 延时函数
{
  unsigned int i =0;
  for (i =0;i < n;i + + )
  delay_1us( ) ;
}
void delay_1ms( void)                    //1ms 延时函数
{
  unsigned int i;
  for (i =0;i < 140;i + + ) ;
}
void delay_nms( unsigned int n)       //n ms 延时函数
{
  unsigned int i =0;
  for (i =0;i < n;i + + )
  delay_1ms( ) ;
}

/ **********************************************************************
函数功能:主程序
入口参数:
出口参数:
 ********************************************************************** /
void main( void)
{
```

```
    DDRB = 0xff;
    PORTB = 0xff;

    while(1)
      {
        RRLAY_On( ) ;
        delay_nms(500);
        RELAY_Off( );
        delay_nms(500);
      }
    }
```

10.6　数码管实验

在一般的人机对话中，输入器件一般都是以按键为主，但输出器件则以数码管或 LCD 为主。在本节的基础实验中先介绍数码管的应用。数码管作为一种应用十分普遍的显示器件，可以在各种各样的设备上见到，例如图 10-12 就是某数字表头显示时的效果图。它很适合用在对价格、亮度等条件比较敏感，同时基本上只要求显示数字量的时候，所以在数据显示、定时控制等场合用得很多。常见的数码管实物如图 10-13 所示。

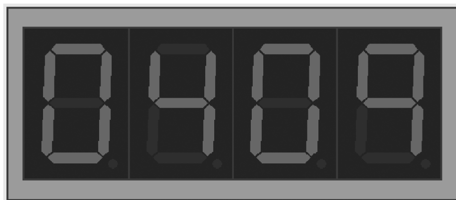


图 10-12　数码管显示效果图

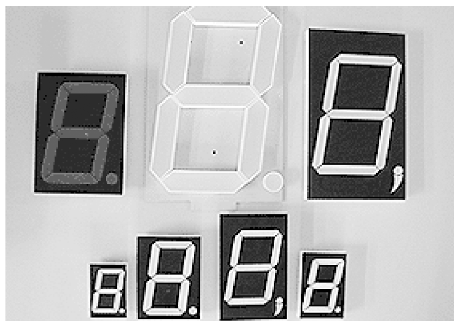


图 10-13　数码管实物图

10.6.1　实例功能

在本节里，将介绍数码管的原理和最简单的使用方法。本例要在 4 个数码管上实现显示数字 1234 的程序。

10.6.2　器件与原理

数码管也叫 LED 数码显示器，其实是由多个 LED 排列封装而成，图 10-13 给出了一些常见的数码管的实物图，其中以上看到的数码管都是由 8 个 LED 组合而成的，当然也有其他类型的，其结构图如图 10-14 所示，其中 7 个发光二极管排列成 8 字形，另外一个则是圆点状的，通常用来作为数据显示时的小数点使用。

由于驱动方式的差异，也就是对应在各个显示段是低电平还是高电平点亮，数码管又分

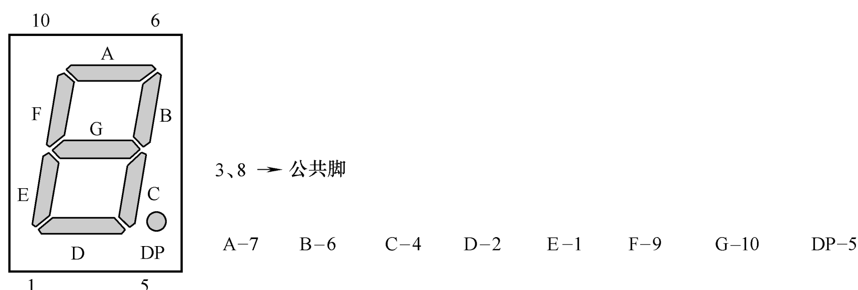


图 10-14　数码管结构图

成两种类型，即共阳极和共阴极数码管。所谓“共阳极”即是 8 个 LED 的阳极连接在一起组成公共端，同理“共阴极”则是 8 个 LED 的阴极连接在一起组成公共端，其内部 LED 的连接方式可以参考图 10-15 所示。

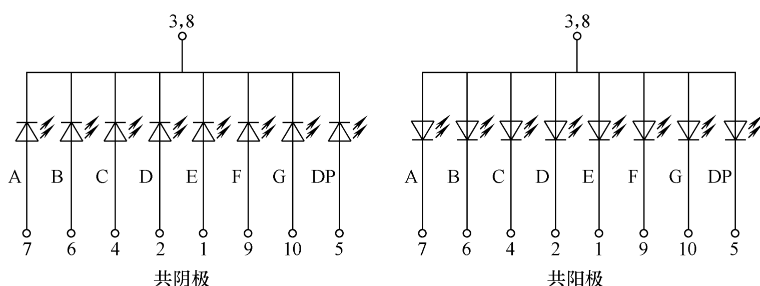


图 10-15　数码管内部结构图

虽然通过上文的原理介绍，对数码管的工作原理已经了解，但当我们拿到一个数码管时要正确地应用它还是一时不知如何下手，比如现在要求数码管显示“5”，需要怎么办呢？首先需要明白一个事情，数码管是不认识“5”的，当然也不认识其他数字，所以千万别说，“给数码管写个‘5’就行了”，数字只是符号，对人来说是这样的，对单片机而言也是，单片机只是通过 LED 是把内部的结果用我们约定的方式显示出来而已，这个“约定”就是数字该如何在 LED 上显示的方法。比如需要显示的数字 0 ~ 9，如图 10-16 所示，并且假设使用共阴极数码管，然后对照图 10-14 和图 10-15 来看看“5”是如何显示出来的。首先对数码管而言，要想显示数字“5”，可以发现有如下一些段是需要点亮的，即 A、C、D、F、G，其他的端口都要置成“1”。如果在程序里，我们通过先查表，然后送出去来实现段码显示的，并且高、低电平刚好对应起来，那么可以列出段码对应关系表，见表 10-3。

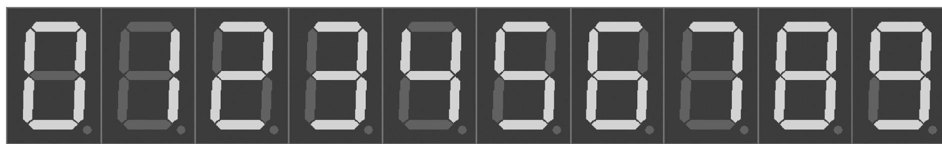


图 10-16　显示数字效果图

表 10-3　数码管显示数字“5”的段码表

段名称	DP	G	F	E	D	C	B	A	对应段码
数字 5	0	1	1	0	1	1	0	1	0x6D

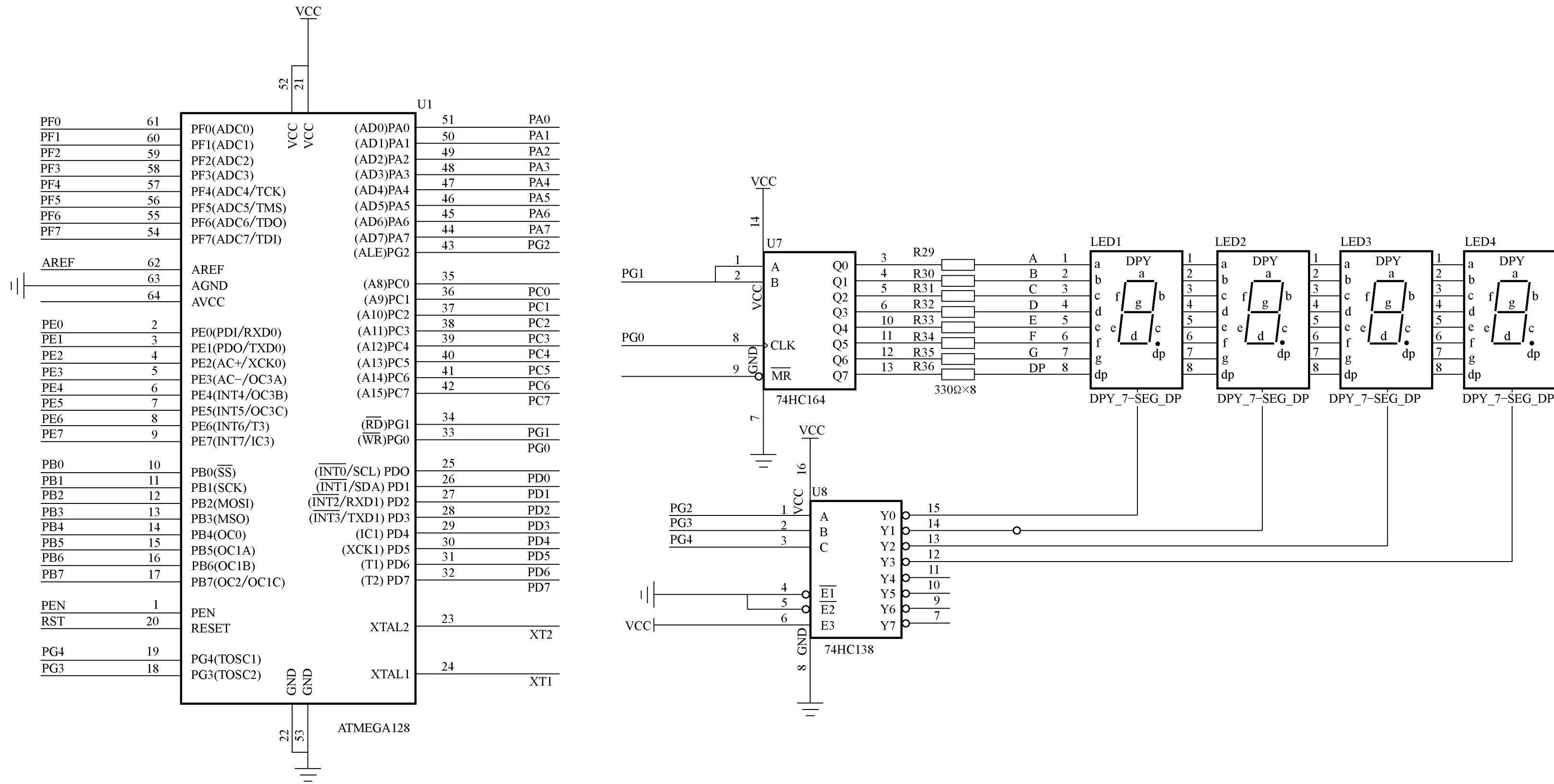


图 10-17　数码管显示原理图

参照上面的过程，可以列出共阴极和共阳极数码管 0 ~ 9 十个数字的段码表，如表 10-4 所示，在不改变硬件对应关系的前提下，段码表可以通用。

表 10-4　共阴极、共阳极数码管段码表

数字	0	1	2	3	4	5	6	7	8	9
共阴极	0x3F	0x06	0x5B	0x4F	0x66	0x6D	0x7D	0x07	0x7F	0x6F
共阳极	0xC0	0xF9	0xA4	0xB0	0x99	0x92	0x82	0xF8	0x80	0x90

现在已经了解了整个显示过程，所以也就有了写程序的思路：程序中应该有一个变量，把变量中的数进行查表得到段码表，把查到的数据送到控制数码管的段码口。

10.6.3　硬件电路（见图 10-17）

10.6.4　程序设计

```
/ ********************************************************************** /
/ * 数码管测试程序                                                      * /
/ * 目标器件:ATmega16                                                   * /
/ * 晶振:RC 1MHz                                                        * /
/ * 编译环境:ICCAVR 6.31A                                               * /
/ ********************************************************************** /

/ ************************** 包含头文件 ************************** /
#include  <iom16v.h>
#include  <macros.h>

/ ***************************** 端口定义 ******************************* /
#define     sclon      PORTG |= BIT(0);        //时钟高
#define     scloff     PORTG &= ~BIT(0);
#define     dion       PORTG |= BIT(1);        //数据高
#define     dioff      PORTG &= ~BIT(1);

/ ********************************* 数码管段码表 ************************** /
extern const unsigned char
tab[] = {0x3f,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};

/ *********************************************************************
函数功能:数据输出程序
入口参数:temp
出口参数:
 ********************************************************************* /
void dataOUT(unsigned char temp)
```

```
{
unsigned char i,temp1;
temp1 = tab[temp];
for(i = 0;i < 8;i++)
{
scloff;
     NOP();
     if((temp1&0x80)!= 0x80)
{ dioff;
        NOP();}
else
{ NOP();
        dion;
        NOP();}
          sclon;
        NOP();
        NOP();
     temp1 <<=1;
        NOP();
        NOP();
        scloff;
        }
}

/***********************************************************************
函数功能:延时程序
入口参数:
出口参数:
***********************************************************************/
void delay(void)
{
   int i;
   for(i=0;i<200;i++);
}

/***********************************************************************
函数功能:显示子程序
入口参数:k
出口参数:
***********************************************************************/
```

```
void display(unsigned int k)
{
    dataOUT(k/1000);
    PORTG &=0x03;
    delay();
    dataOUT(k/100%10);
    PORTG |=0x04;
    delay();
    dataOUT(k/10%10);
    PORTG &=0x0B;
    PORTG |=0x08;
    delay();
    dataOUT(k%10);
    PORTG |=0x0C;
    delay();
}

/*********************************************************************
函数功能:主程序
入口参数:
出口参数:
*********************************************************************/
void main(void)
{
DDRG=0xff;
while(1)
   {
   delay();
      display(1234);
      delay();
   }
}
```

10.7　串行口实验

单片机除了需要控制外围元器件完成特定的功能外，在很多应用中还要完成单片机和单片机之间、单片机和外围元器件之间，以及单片机和微机之间的数据交换和指令的传输，这就是单片机的通信。单片机的通信方式可以分为并行通信和串行通信。并行方式传送一个字节的数据至少需要 8 条数据线。一般来讲单片机与打印机等外围设备连接时，除 8 条数据线外，还要状态、应答等控制线，当传送距离过远时电线要求过多，成本会增加很多。单片机

的串行通信方法较为多样，传统的串行通信方式是通过单片机自带的串行口进行 RS232 方式的通信。串行通信是以一位数据线传送数据的位信号，即使加上几条通信联络控制线，也比并行通信用的线少。因此，串行通信适合远距离数据传送，如大型主机与其远程终端之间，处于两地的计算机之间，采用串行通信就非常经济。

10.7.1　实例功能

本例将介绍用单片机的串行口将 PC 端发送过来的数据接收并返回给 PC。ATmega16 自带有两个串行口，两个串口由不同的寄存器控制。下文以 UART0 为例，通过这个实例可以掌握串行口的收发应用。

与 USART 相关的有以下几个寄存器：数据寄存器——UDR、控制和状态寄存器 A——UCSRA、控制和状态寄存器 B——UCSRB、控制和状态寄存器 C——UCSRC、波特率寄存器——UBRRH 和 UBRRL。下面简要回顾一下这几个寄存器的功能。

1. 数据寄存器——UDR

USART 发送数据缓冲寄存器和 USART 接收数据缓冲寄存器共享相同的 I/O 地址，称为 USART 数据寄存器或 UDR。将数据写入 UDR 时实际操作的是发送数据缓冲器存器（TXB），读 UDR 时实际返回的是接收数据缓冲寄存器（RXB）的内容。

bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDR 读
	TXB[7:0]								UDR 写
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初值	0	0	0	0	0	0	0	0	

在 5、6、7bit 字长模式下，未使用的高位被发送器忽略，而接收器则将它们设置为 0。只有当 UCSRA 寄存器的 UDRE 标志置位后才可以对发送缓冲器进行写操作。如果 UDRE 没有置位，那么写入 UDR 的数据会被 USART 发送器忽略。当数据写入发送缓冲器后，若移位寄存器为空，发送器将把数据加载到发送移位寄存器。然后数据串行地从 TxD 引脚输出。接收缓冲器包括一个两级 FIFO，一旦接收缓冲器被寻址 FIFO 就会改变它的状态。因此不要对这一存储单元使用读-修改-写指令（SBI 和 CBI）。使用位查询指令（SBIC 和 SBIS）时也要小心，因为这也有可能改变 FIFO 的状态。

2. 控制和状态寄存器 A——UCSRA

bit	7	6	5	4	3	2	1	0
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
读/写	R	R/W	R	R	R	R	R/W	R/W
初值	0	0	1	0	0	0	0	0

（1）bit 7-RXC：USART 接收结束

接收缓冲器中有未读出的数据时 RXC 置位，否则清零。接收器禁止时，接收缓冲器被刷新，导致 RXC 清零。RXC 标志可用来产生接收结束中断（见对 RXCIE 位的描述）。

（2）bit 6-TXC：USART 发送结束

发送移位缓冲器中的数据被送出，且当发送缓冲器（UDR）为空时 TXC 置位。执行发送结束中断时 TXC 标志自动清零，也可以通过写 1 进行清除操作。TXC 标志可用来产生发送结束中断（见对 TXCIE 位的描述）。

（3）bit 5-UDRE：USART 数据寄存器空

UDRE 标志指出发送缓冲器（UDR）是否准备好接收新数据。UDRE 为 1 说明缓冲器为空，已准备好进行数据接收。UDRE 标志可用来产生数据寄存器空中断（见对 UDRIE 位的描述）。复位后 UDRE 置位，表明发送器已经就绪。

（4）bit 4-FE：帧错误

如果接收缓冲器接收到的下一个字符有帧错误，即接收缓冲器中的下一个字符的第一个停止位为 0，那么 FEn 置位。这一位一直有效直到接收缓冲器（UDR）被读取。当接收到的停止位为 1 时，FE 标志为 0。对 UCSRA 进行写入时，这一位要写 0。

（5）bit 3-DOR：数据过速

数据过速时 DOR 置位。当接收缓冲器满（包含了两个数据），接收移位寄存器又有数据，若此时检测到一个新的起始位，数据溢出就产生了。这一位一直有效，直到接收缓冲器（UDR）被读取。对 UCSRA 进行写入时，这一位要写 0。

（6）bit 2-PE：奇偶校验错误

当奇偶校验使能（UPM1 =1），且接收缓冲器中所接收到的下一个字符有奇偶校验错误时，UPE 置位。这一位一直有效，直到接收缓冲器（UDR）被读取。对 UCSRA 进行写入时，这一位要写 0。

（7）bit 1-U2X：倍速发送

这一位仅对异步操作有影响。使用同步操作时将此位清零。此位置 1 可将波特率分频因子从 16 降到 8，从而有效地将异步通信模式的传输速率加倍。

（8）bit 0-MPCM：多处理器通信模式

设置此位将启动多处理器通信模式。MPCM 置位后，USART 接收器接收到的那些不包含地址信息的输入帧都将被忽略。发送器不受 MPCM 设置的影响。

3. 控制和状态寄存器 B——UCSRB

bit	7	6	5	4	3	2	1	0
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W
初值	0	0	0	0	0	0	0	0

（1）bit 7-RXCIE：接收结束中断使能

置位后使能 RXC 中断。当 RXCIE 为 1，全局中断标志位 SREG 置位，UCSRA 寄存器的 RXC 亦为 1 时可以产生 USART 接收结束中断。

（2）bit 6-TXCIE：发送结束中断使能

置位后使能 TXC 中断。当 TXCIE 为 1，全局中断标志位 SREG 置位，UCSRA 寄存器的 TXC 亦为 1 时可以产生 USART 发送结束中断。

（3）bit 5-UDRIE：USART 数据寄存器空中断使能

置位后使能 UDREn 中断。当 UDRIE 为 1，全局中断标志位 SREG 置位，UCSRA 寄存器的 UDRE 亦为 1 时可以产生 USART 数据寄存器空中断。

（4）bit 4-RXEN：接收使能

置位后将启动 USART 接收器。RxD 引脚的通用端口功能被 USART 功能所取代。禁止接收器将刷新接收缓冲器，并使 FE、DOR 及 PE 标志无效。

（5）bit 3-TXEN：发送使能

置位后将启动 USART 发送器。TxD 引脚的通用端口功能被 USART 功能所取代。TXEN 清零后，只有等到所有的数据发送完成后发送器才能够真正禁止，即发送移位寄存器与发送缓冲寄存器中没有要传送的数据。发送器禁止后，TxD 引脚恢复其通用 I/O 功能。

（6）bit 2-UCSZ2：字符长度

UCSZ2 与 UCSRC 寄存器的 UCSZ1：0 结合在一起可以设置数据帧所包含的数据位数（字符长度）。

（7）bit 1-RXB8：接收数据位 8

对 9 位串行帧进行操作时，RXB8 是第 9 个数据位。读取 UDR 包含的低位数据之前首先要读取 RXB8。

（8）bit 0-TXB8：发送数据位 8

对 9 位串行帧进行操作时，TXB8 是第 9 个数据位。写 UDR 之前首先要对它进行写操作。

4. 控制和状态寄存器 C——UCSRC

bit	7	6	5	4	3	2	1	0
	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCOPL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	1	0	0	0	0	1	1	0

（1）bit 7-URSEL：寄存器选择

通过该位选择访问 UCSRC 寄存器或 UBRRH 寄存器。当读 UCSRC 时，该位为 1 ；当写 UCSRC 时，URSEL 为 1。

（2）bit 6-UMSEL：USART 模式选择

通过这一位来选择同步或异步工作模式。

UMSEL＝0：异步操作

UMSEL＝1：同步操作

（3）bit 5∶4-UPM1∶0：奇偶校验模式

这两位设置奇偶校验的模式并使能奇偶校验。如果使能奇偶校验，那么在发送数据时，发送器都会自动产生并发送奇偶校验位。对每一个接收到的数据，接收器都会产生一个奇偶值，并与 UPM0 所设置的值进行比较。如果不匹配，那么就将 UCSRA 中的 UPE 置位。

UPM1	UPM0	奇偶模式
0	0	禁止
0	1	保留
1	0	偶校验
1	1	奇校验

（4）bit 3-USBS：停止位选择

通过这一位可以设置停止位的位数。接收器忽略这一位的设置。

USBS＝0：1 位停止位

USBS＝1：2 位停止位

（5）bit 2∶1-UCSZ1∶0：字符长度

UCSZ1：0 与 UCSRB 寄存器的 UCSZ2 结合在一起可以设置数据帧包含的数据位数（字符长度）。

UCSZ2	UCSZ1	UCSZ0	字符长度(位)
0	0	0	5
0	0	1	6
0	1	0	7
0	1	1	8
1	0	0	保留
1	0	1	保留
1	1	0	保留
1	1	1	9

（6）bit 0-UCPOL：时钟极性

这一位仅用于同步工作模式。使用异步模式时，将这一位清零。UCPOL 设置了输出数据的改变和输入数据采样，以及同步时钟 XCK 之间的关系。

UCPOL	发送数据的改变(TxD 引脚的输出)	接收数据的采样(RxD 引脚的输入)
0	XCK 上升沿	XCK 下降沿
1	XCK 下降沿	XCK 上升沿

5. 波特率寄存器——UBRRH 和 UBRRL

bit	15	14	13	12	11	10	9	8	UBRRH
	URSEL				UBRR[11:8]				
	UBRR[7: 0]								UBRRL
	7	6	5	4	3	2	1	0	
读/写	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初值	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

（1）bit 15-URSEL：寄存器选择

通过该位选择访问 UCSRC 寄存器或 UBRRH 寄存器。当读 UBRRH 时，该位为 0；当写 UBRRH 时，URSEL 为 0。

（2）bit 14:12-保留

这些位是为以后的使用而保留的。为了与以后的器件兼容，写 UBRRnH 时将这些位清零。

（3）bit 11:0-UBRR11:0：USART 波特率寄存器

这个 12 位的寄存器包含了 USART 的波特率信息。其中 UBRRH 包含了 USART 波特率高 4 位，UBRRL 包含了低 8 位。波特率的改变将造成正在进行的数据传输受到破坏。写 UBRRL 将立即更新波特率分频器。

通过以上寄存器内容的回顾，接下来就可以进行串行数据通信实验。

10.7.2　硬件电路

从本书第 2 章可知，ATmega16 单片机有一个全双工的串行通信口，所以单片机和计算机之间可以方便地进行串口通信。进行串行通信时要满足一定的条件，比如计算机的串口是

RS232 电平的，而单片机的串口是 TTL 电平的，两者之间必须有一个电平转换电路，我们采用了专用芯片 MAX232 进行转换，虽然也可以用几个晶体管进行电平转换，但是还是用专用芯片更简单可靠。MAX232 芯片是 MAXIM 公司生产的、包含两路接收器和驱动器的 IC 芯片，外部引脚和内部电路如图 10-18 所示。

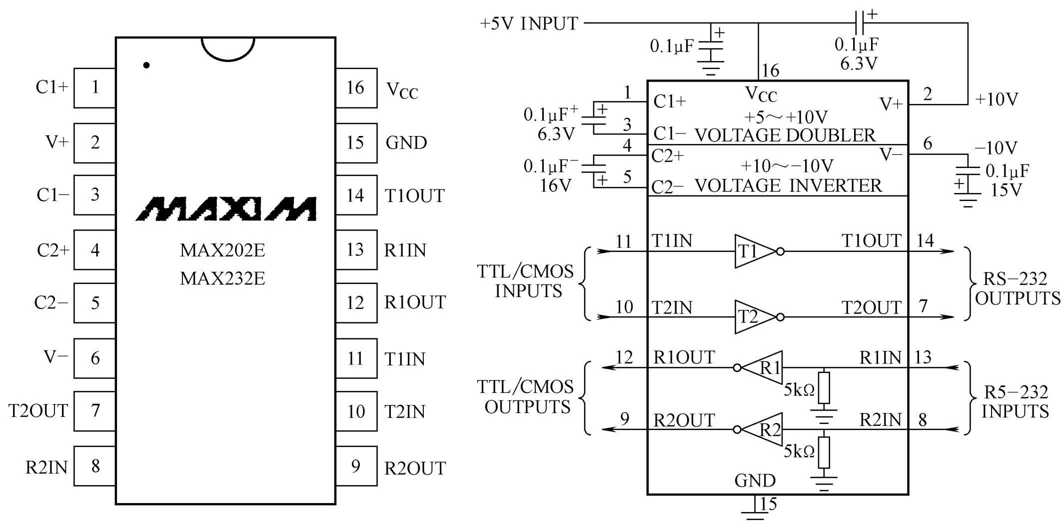


图 10-18　MAX232 引脚及内部框图

在实际应用中一般采用如图 10-19 所示硬件电路图，这是最简单的连接方法，但是对我们来说已经足够使用了。

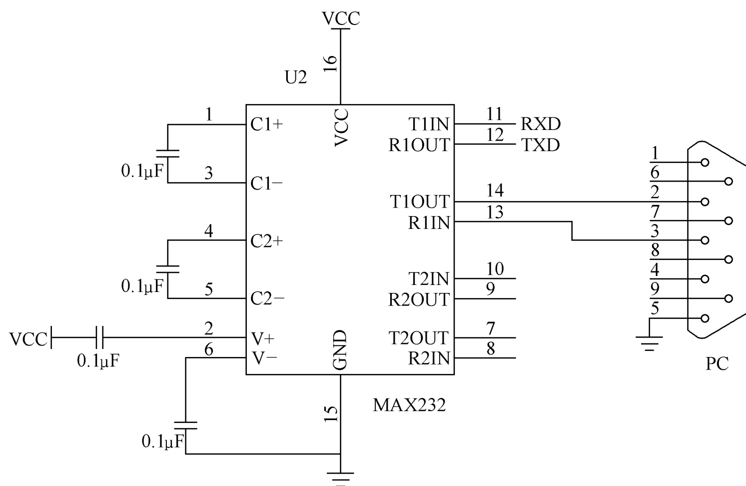


图 10-19　电平转换原理图

为了能够在计算机端看到单片机发出的数据，必须借助一个 Windows 软件进行观察，这里推荐一个免费的计算机串口调试软件——串口调试助手。

此软件如图 10-20 所示，可设定串口号、波特率、校验位等参数，非常实用。在实际应用中一定要保证上位机设置与单片机相统一，否则数据将会出错。本例硬件电路图如图 10-21 所示。

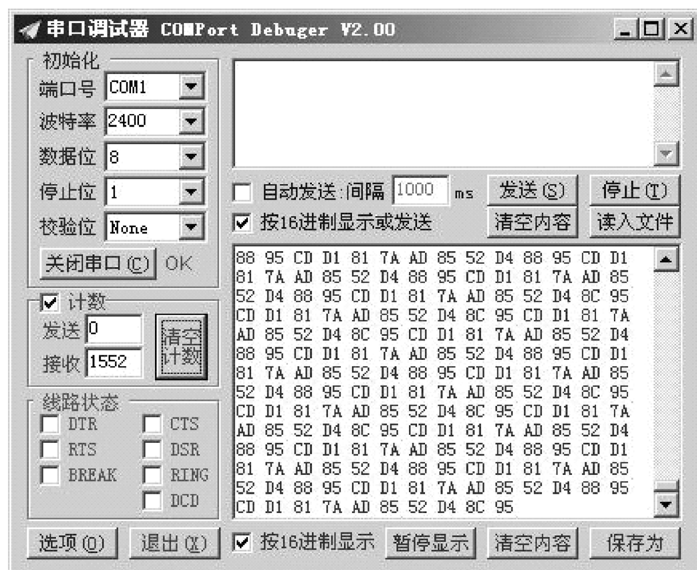


图 10-20　串口调试助手界面图

10.7.3　程序设计

```
/ ************************************************************************ /
/ * 串口测试程序                                                           * /
/ * 目标器件:ATmega16                                                      * /
/ * 晶振:RC 1MHz                                                           * /
/ * 编译环境:ICCAVR 6.31A                                                  * /
/ ************************************************************************ /

/ ************************** 包含头文件 ************************** /
#include <iom16v.h>
#include <macros.h>

/ ******************************** 宏定义 ******************************** /
#define fosc 1000000    //晶振 1MHz
#define baud 2400       //波特率

/ ***********************************************************************
函数功能:uart 初始化程序
入口参数:
出口参数:
************************************************************************ /
void uart_init(void)
{
```

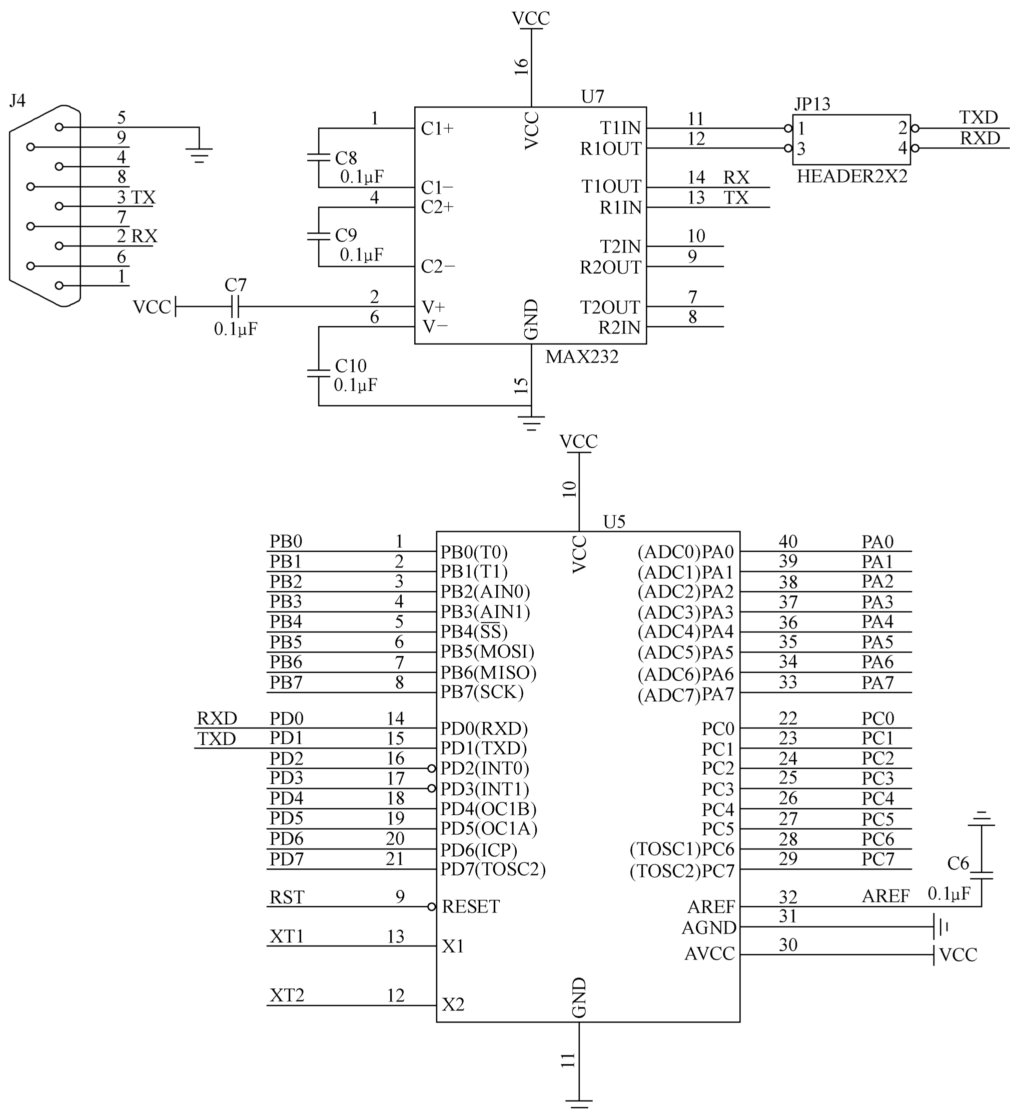


图 10-21　串行通信硬件电路图

```
UCSRB  = 0x00;
UCSRA  = 0x00;
UCSRC  =(1 < <UCSZ1)|(1 < <UCSZ0);//8 位
UBRRL =(fosc/16/(baud +1))%256;
UBRRH =(fosc/16/(baud +1))/256;
UCSRB  =(1 < <RXEN)|(1 < <TXEN); //发送使能,接收使能
}

/ ***********************************************************************
函数功能:uart 发送单字节数据
入口参数:c
```

```
出口参数:
************************************************************************/
void putchar(unsigned char c)
{
     while (!(UCSRA&(1<<UDRE)));//检测数据存储器是否为空
     UDR=c;
}

/************************************************************************
函数功能:uart 接收单字节数据
入口参数:
出口参数:
************************************************************************/
unsigned char getchar(void)
{
     while(!(UCSRA& (1<<RXC)));//检测接收是否结束
     return UDR;
}
/************************************************************************
函数功能:uart 发送字符串数据
入口参数:*s
出口参数:
************************************************************************/

void puts(char *s)
{
   while (*s)
     {
     putchar(*s);
     s++;
     }
   putchar(0x0a);
   putchar(0x0d);
}

/************************************************************************
函数功能:主程序
入口参数:
出口参数:
************************************************************************/
```

```
void main(void)
{
    unsigned char i;

    uart_init();
    while(1)
    {
    puts("HELLO!");
    puts("test ok!");
  }
}
```

第 11 章　ATmega16 高级应用实例

11.1　矩阵键盘应用实例

在前面章节中本书介绍了独立按键的应用，独立按键具有编程简单但占用 I/O 口资源的特点，不适合在按键较多的场合应用。在实际应用中经常要用到输入数字、字母等功能，如电子密码锁、电话机键盘等一般都至少有 12～16 个按键，在这种情况下如果用独立按键的话显然太浪费 I/O 口资源，为此就引入了矩阵键盘的应用。

11.1.1　矩阵键盘简介

矩阵键盘又称行列键盘，它是用 4 条 I/O 线作为行线，4 条 I/O 线作为列线组成的键盘。在行线和列线的每个交叉点上设置一个按键。这样键盘上按键的个数就为 4×4 个。这种行列式键盘结构能有效地提高单片机系统中 I/O 口的利用率。

11.1.2　矩阵键盘的工作原理

最常见的键盘布局如图 11-1 所示。一般由 16 个按键组成，在单片机中正好可以用一个 P 口实现 16 个按键功能，这也是在单片机系统中最常用的形式，4×4 矩阵键盘的内部电路如图 11-2 所示。

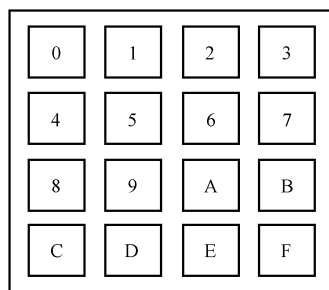


图 11-1　矩阵键盘布局图

当无按键闭合时，PD0～PD3 与 PD4～PD7 之间开路。当有按键闭合时，与闭合键相连的两条 I/O 口线之间短路。判断有无按键按下的方法是：第一步，置列线 PD4～PD7 为输入状态，从行线 PD0～PD3 输出低电平，读入列线数据，若某一列线为低电平，则该列线上有按键闭合。第二步，行线轮流输出低电平，从列线 PD4～PD7 读入数据，若有某一列为低电平，则对应行线上有按键按下。综合一二两步的结果，可确定按键编号。但是按键闭合一次只能进行一次键功能操作，因此须等到按键释放后，再进行键功能操作，否则按一次键，有可能会连续多次进行同样的键操作。

11.1.3　矩阵键盘软硬件设计实例

本节利用配套开发板为硬件平台，介绍矩阵式键盘的编程方法。本实验通过按下相应键后在一位数码管上显示出键值。0～16 个键分别对应显示 0～F。

1. 硬件原理图

本实验可以直接在配套开发板上完成，其硬件原理图如图 11-3 所示。

根据图 11-3，键盘扫描方法是：行线 PD0～PD3 为输出线，列线 PD4～PD7 为输入线。一开始单片机将行线（PD0～PD3）全部输出低电平，此时读入列线数据，若列线全为高电

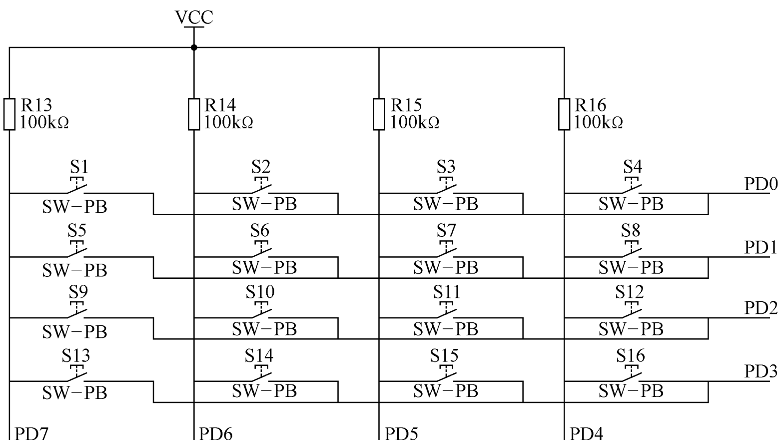


图 11-2　矩阵键盘内部电路图

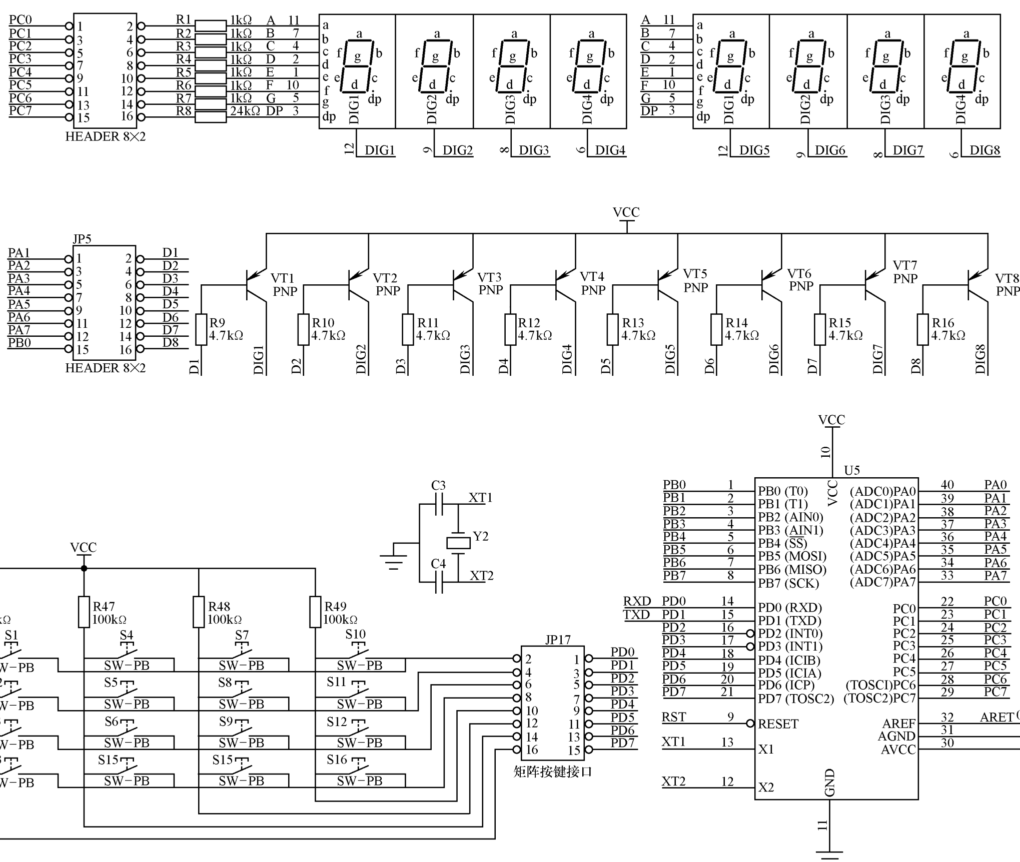


图 11-3　硬件原理图

平则没有键按下，当列线有出现低电平时调用延时程序，以此来去除按键抖动。延时完成后再判断是否有低电平，如果此时读入列线数据还是有低电平，则说明确实有键按下。最后一步确定键值。现在以第 2 行的 S5 键为例，若按下 S5 后应该怎么得到这个键值呢？当判断确实有键按下之后，行线轮流输出低电平，根据读入列线的数据可以确定键值。首先，单片机将 PD0 输出为低电平，其他 PD1 ~ PD3 输出高电平，此时读取列线的数据全为高电平，说明

没有在第一行有键按下；其次，单片机将 PD1 输出低电平，其他 PD0、PD2、PD3 仍为高电平，此时再来读取列线数据，发现列线读到的数据有低电平，数值为 1011（0x0B），如果键盘布局已经确定，那么 0x0B 就代表 S5 的值了。转到 S5 键功能处理子程序就可以达到目的。

2. 软件设计

```
/*************************************************************/
/*矩阵键盘测试程序                                            */
/*目标器件:ATmega16                                          */
/*晶振:RC 1MHz                                               */
/*编译环境:ICCAVR 6.31A                                      */
/*************************************************************/

/*********************包含头文件*****************************/
#include <iom16v.h>
#include <macros.h>

/**********************数码管段码表************************
extern const unsigned char
tab[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,
/*     0    1    2    3    4 5    6    7  8    9   */
0x88,0x83,0xC6,0xA1,0x86,0x8E};
/*       a    b    c    d    e    f   */
*************************************************************/
函数功能:1μs 延时程序
入口参数:
出口参数:
*************************************************************/
void delay_1us(void)                     //1μs 延时函数
{
    asm("nop");
}

/************************************************************
函数功能:Nμs 延时程序
入口参数:n
出口参数:
*************************************************************/

void delay_nus(unsigned int n)           //Nμs 延时函数
{
    unsigned int i = 0;
```

```
    for(i=0;i<n;i++)
    delay_1us();
}

/**********************************************************
函数功能:1ms 延时程序
入口参数:n
出口参数:
**********************************************************/
void delay_1ms(void)                    //1ms 延时函数
{
    unsigned int i;
    for(i=0;i<140;i++);
}

/**********************************************************
函数功能:nmS 延时程序
入口参数:n
出口参数:
**********************************************************/
void delay_nms(unsigned int n)          //Nms 延时函数
{
    unsigned int i=0;
    for(i=0;i<n;i++)
    delay_1ms();
}

/**********************************************************
函数功能:延时程序
入口参数:
出口参数:
**********************************************************/
void delay(void)
{
    int i;
    for(i=0;i<200;i++);
}

/**********************************************************
函数功能:显示子程序
```

```
入口参数:k
出口参数:
*************************************************************/
void display(unsigned char k)
{
    PORTC = tab[k/10];
    PORTA = 0xef;
    delay_nms(2);
    PORTC = tab[k%10];
    PORTA = 0xf7;
    delay_nms(2);
}

/*************************************************************
函数功能:主程序
入口参数:
出口参数:
*************************************************************/
void main(void)
{
    int keyvalue = 0;
    int j;

    DDRC = 0xff;
    DDRA = 0xff;
    DDRD = 0x0f;
    while(1)
    {
      //扫描第 1 行
      PORTD = 0x0E;
      for(j = 0;j < 5;j++);
      if((PIND&0xF0)!= 0xF0)
      {
        delay();
        if((PIND&0xF0)!= 0xF0)
        {
          if((PIND&0x10) == 0)
            keyvalue = 4;
          if((PIND&0x20) == 0)
            keyvalue = 3;
```

```
        if((PIND&0x40) = =0)
          keyvalue =2;
        if((PIND&0x80) = =0)
          keyvalue =1;
        NOP();
      }
    }
    //扫描第 2 行
    PORTD =0x0D;
    for(j =0;j <5;j + +);
    if((PIND&0xF0)!  =0xF0)
    {
      delay();
      if((PIND&0xF0)!  =0xF0)
      {
        if((PIND&0x10) = =0)
          keyvalue =8;
        if((PIND&0x20) = =0)
          keyvalue =7;
        if((PIND&0x40) = =0)
          keyvalue =6;
        if((PIND&0x80) = =0)
          keyvalue =5;
        NOP();
      }
    }
    //扫描第 3 行
    PORTD =0x0B;
    for(j =0;j <5;j + +);
    if((PIND&0xF0)!  =0xF0)
    {
      delay();
      if((PIND&0xF0)!  =0xF0)
      {
        if((PIND&0x10) = =0)
          keyvalue =12;
        if((PIND&0x20) = =0)
          keyvalue =11;
        if((PIND&0x40) = =0)
          keyvalue =10;
```

```
            if((PIND&0x80) = =0)
              keyvalue =9;
            NOP();
          }
        }
        //扫描第 4 行
        PORTD =0x07;
        for(j =0;j <5;j + +);
        if((PIND&0xF0)!  =0xF0)
        {
          delay();
          if((PIND&0xF0)!  =0xF0)
          {
            if((PIND&0x10) = =0)
              keyvalue =16;
            if((PIND&0x20) = =0)
              keyvalue =15;
            if((PIND&0x40) = =0)
              keyvalue =14;
            if((PIND&0x80) = =0)
              keyvalue =13;
            NOP();
          }
        }
      display(keyvalue);
      }
    }
```

11.2　步进电动机应用实例

11.2.1　步进电动机简介

步进电动机是一种将电脉冲转化为角位移的执行机构。当步进驱动器接收到一个脉冲信号，它就驱动步进电动机就按设定的方向转动一个固定的角度（步距角）。通过控制脉冲个数来控制角位移量，可以达到准确定位；同时可以通过控制脉冲频率来控制电动机转动的速度和加速度，达到调速的目的；可以通过改变各相的通电顺序，控制步进电动机的转动方向。

1. 步进电动机的分类与结构

步进电动机可分为 3 类：

1）永磁式步进电动机。永磁式步进电动机（Permanent Magnet，PM），其转子是用永磁

材料制成的，转子本身就是一个磁源。它的输出转矩大，动态性能好。转子的极数与定子的极数相同，所以步距角一般较大。

2）反应式步进电动机。反应式步进电动机（Variable Reluctance，VR），其转子是用软磁材料制成的，转子中没有绕组。它的结构简单，成本低，步距角可以做得很小，但动态性能很差。

3）混合式步进电动机。混合式步进电动机（Hybrid，HB）综合了反应式和永磁式的优点，它的输出转矩大，动态性能好，步距角小，但结构复杂，成本较高。这种步进电机的应用最为广泛。

步进电动机的内部模型如图 11-4 所示。

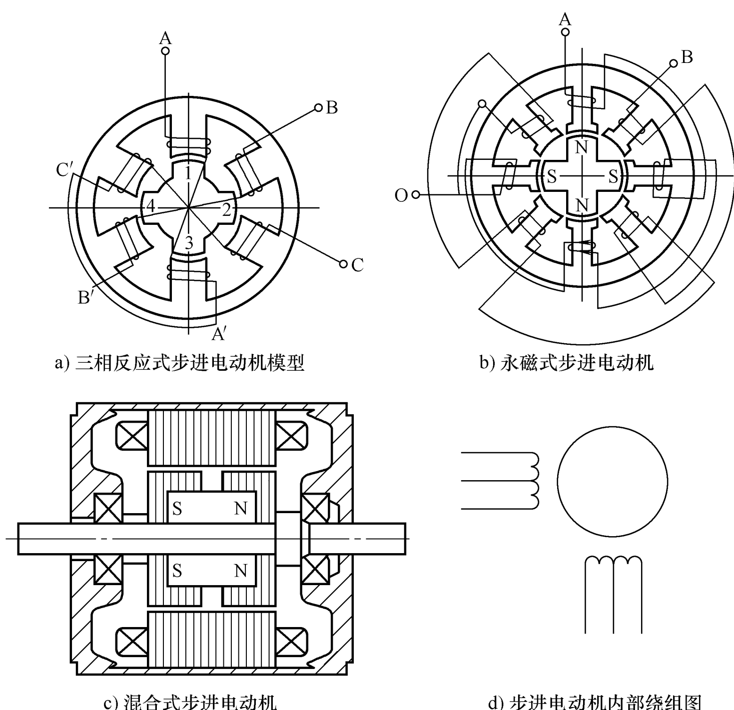


图 11-4　主流步进电动机的内部模型图

2. 步进电动机的基本参数

在应用选型时可以根据电动机的不同参数来决定应用范围，一般步进电动机主要有以下几个参数：步距角、相数、保持转矩、静转矩、拍数等。

（1）电动机固有步距角

电动机固有步距角表示控制系统每发一个步进脉冲信号，电动机所转动的角度。一般电动机出厂时都给出了一个步距角的值，如 86BYG250A 型电动机给出的值为 0.9°/1.8°（表示半步工作时为0.9°，整步工作时为1.8°），这个步距角可以称之为电动机“固有步距角”，它不一定是电动机实际工作时的真正步距角，真正的步距角还和驱动器有关。

电动机转子转过的角位移用 θ 表示。$\theta = 360°/($转子齿数 $J\times$ 运行拍数$)$，以常规二、四相，转子齿为 50 齿电动机为例。四拍运行时步距角为 $\theta = 360°/(50\times4) = 1.8°$（俗称整步），八拍运行时步距角为 $\theta = 360°/(50\times8) = 0.9°$（俗称半步）。

（2）步进电动机的相数

步进电动机的相数是指电动机内部产生不同对极 N、S 磁场的励磁线圈对数，常用 m 表示。目前常用的有二相、三相、四相、五相步进电动机。电动机相数不同，其步距角也不同，一般二相电动机的步距角为 0.9°/1.8°、三相的为 0.75°/1.5°、五相的为 0.36°/0.72° 。在没有细分驱动器时，用户主要靠选择不同相数的步进电动机来满足自己步距角的要求。

（3）保持转矩（HOLDING TORQUE）

保持转矩是指步进电动机通电但没有转动时，定子锁住转子的力矩。它是步进电动机最重要的参数之一，通常步进电动机在低速时的力矩接近保持转矩。由于步进电动机的输出力矩随速度的增大而不断衰减，输出功率也随速度的增大而变化，所以保持转矩就成为了衡量步进电动机最重要的参数之一。比如，当人们说 2N · m 的步进电动机，在没有特殊说明的情况下是指保持转矩为 2N · m 的步进电动机。

（4）DETENT TORQUE

DETENT TORQUE 是指步进电动机没有通电的情况下，电动机转子自身的锁定力矩。DETENT TORQUE 在国内有时也称为定位转矩。由于反应式步进电动机的转子不是永磁材料，所以它没有 DETENT TORQUE。

（5）静转矩

静转矩表示电动机在额定静态电作用下，电动机不作旋转运动时，电动机转轴的锁定力矩。此力矩是衡量电动机体积（几何尺寸）的标准，与驱动电压及驱动电源等无关。

（6）拍数

步进电动机拍数是指完成一个磁场周期性变化所需脉冲数或导电状态，用 n 表示，或指电动机转过一个齿距角所需脉冲数，以四相电动机为例，有四相四拍运行方式即 AB-BC-CD-DA-AB，四相八拍运行方式即 A-AB-B-BC-C-CD-D-DA-A。

除了以上常用的参数外，步进电动机还有很多其他参数，如步距角精度、失步、失调角、最大空载起动频率等，可以查阅相关技术资料，在此不作介绍。

3. 步进电动机的特性

步进电动机有如下特点：

1）步进电动机的角位移与输入脉冲严格成正比，因此，它没有累计误差，具有良好的跟随性。

2）步进电动机的动态响应快，易于起停、正反转及变速。

3）由步进电动机与驱动电路组成的开环数控系统，既非常简单、廉价，又非常可靠。同时，它也可以与角度反馈环节组成高性能的闭环数控系统。

4）速度可在相当宽的范围内平滑调节，低速下仍能保证获得较大转矩，因此，一般可以不用减速装置而直接驱动负载。

5）步进电动机只能通过脉冲电源供电才能运行，它不能直接使用交流电源和直流电源。

6）步进电动机存在振荡和失步现象，必须对控制系统和机械负载采取相应的措施。

7）步进电动机自身的噪声和振动较大，带惯性负载的能力较差。

4. 反应式步进电动机的结构

本节以反应式步进电动机为例，介绍其基本原理与应用方法。三相反应式步进电动机结构图如图 11-5 所示。从图中可以看出，它分成转子和定子两部分。定子是由硅钢片叠成的。定子上有 6 个磁极（大极），每 2 个相对的磁极（N、S 极）组成一对，共 3 对。每对磁极

都缠有同一绕组，也即形成一相，这样 3 对磁极有 3 个绕组，形成三相。类似地，四相步进电动机有 4 对磁极、4 相绕组；五相步进电动机有 5 对磁极、5 相绕组……每个磁极的内表面都分布着多个小齿，它们大小相同，间距相同。

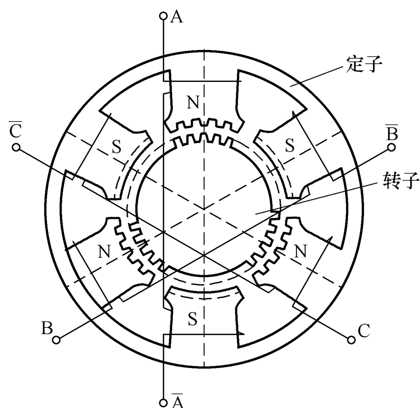


图 11-5　三相反应式步进电动机结构图

转子是由软磁材料制成的，其外表面也均匀分布着小齿，这些小齿与定子磁极上的小齿的齿距相同，形状相似。

由于小齿的齿距相同，所以不管是定子还是转子，其步距角都可计算如下：

$$\theta_Z = 2\pi / Z$$

式中　Z——转子齿数。

例如，如果转子的齿数为 40，则齿距角为 $\theta_Z = 2\pi/40 = 9°$。

把定子小齿与转子小齿对齐的状态称为对齿（见图 11-6a），把定子小齿与转子小齿不对齐的状态称为错齿（见图 11-6b）。错齿的存在是步进电动机能够旋转的前提条件，所以在步进电动机的结构中必须保证有错齿存在，也就是说，当某一相处于对齿状态时，其他相必须处于错齿状态。

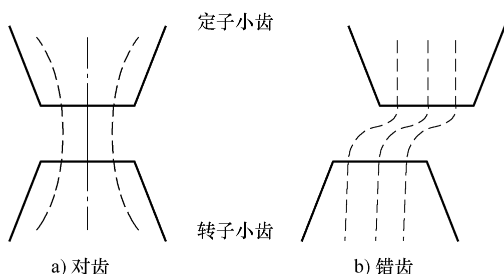


图 11-6　定子小齿与转子小齿间的磁导现象

例如，如果转子有 40 个齿，则转子的齿距角为 9°，因为定子的齿距角与转子相同，定子的齿距角也是 9°。所不同的是，转子的齿是圆周分布的，而定子的齿只分布在磁极上，属于不完全齿。当某一相处于对齿状态时，该相磁极上定子的所有小齿都与转子上的小齿对齐，如图 11-7 所示。

三相步进电动机的每一相磁极在空间上相差 120°。假如当前 A 相处于对齿状态，以 A 相位置作为参考点，B 相与 A 相相差 120°，C 相与 A 相相差 240°。下面可以计算当 A 相处于对齿状态时，B、C 两相的错齿程度如图 11-8 所示。

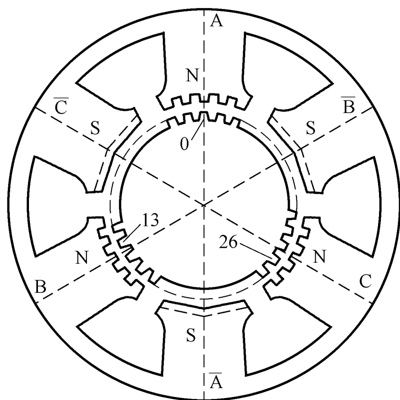


图 11-7　A 相对齿时定转子齿的位置关系

将 A 相磁极中心线看成 0°，在 0°处的转子齿为 0 号齿，则在 120°处的 B 相磁极中心线上对应的转子齿号为 120°/9° = 13（1/3），即 B 相磁极中心线处于转子第 13 号齿再过 1/3 个齿距角的地方。这说明 B 相错了 1/3 个齿距角，即错齿 3°。

同理，与 A 相相差 240°的 C 相磁极中心线上对应的齿号为 240°/9° = 26（2/3），即 C 相磁极中心线处于转子第 26 号齿再过 2/3 齿距角的地方。这说明 C 相错齿 6°。

5. 反应式步进电动机的工作原理

（1）反应式步进电动机的步进原理

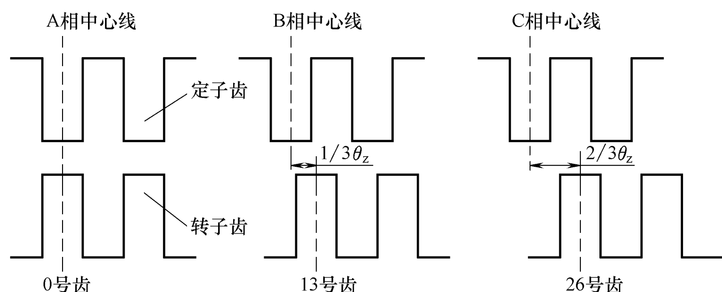


图 11-8　0、13、26 号转子齿与定子齿的位置关系

如果给处于错齿状态的相通电，则转子在电磁力的作用下，将向磁导率最大（或磁阻最小）的位置转动，即向趋于对齿的状态转动。步进电动机就是基于这一原理转动的。

步进电动机的步进过程可描述如下：

如图 11-9 所示，当开关 K_A 合上时，A 相绕组通电，使 A 相磁场建立。当 A 相定子磁极上的齿与转子的齿形成对齿时，B 相、C 相上的齿与转子的齿形成错齿。

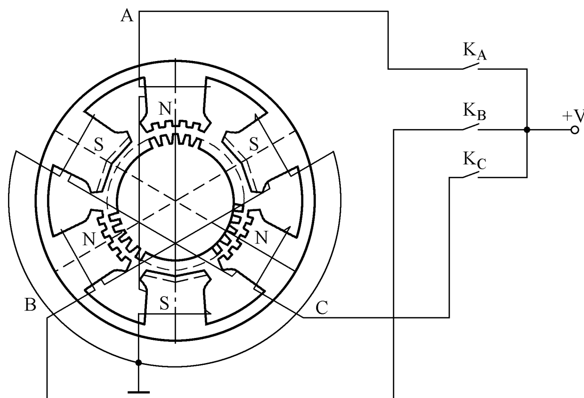


图 11-9　步进电动机的步进原理

将 A 相断电，同时将 K_B 合上，使处于错 1/3 个齿距角的 B 相通电，并建立磁场。转子在电磁力的作用下，向与 B 相成对齿的位置转动。其结果是：转子转动了 1/3 个齿距角；B 相与转子形成对齿；C 相与转子错 1/3 个齿距角；A 相与转子错 2/3 个齿距角。

同理，在 B 相断电的同时，合开关 K_C 给 C 相通电建立磁场，转子又转动了 1/3 个齿距角，与 C 相形成对齿，并且 A 相与转子错 1/3 个齿距角；B 相与转子错 2/3 个齿距角。

当 C 相断电，再给 A 相通电时，转子又转动了 1/3 个齿距角，与 A 相形成对齿，与 B、C 两相形成错齿。至此，所有的状态与最初时一样，只不过转子累计转过了一个齿距角。

可见，由于按 A→B→C→A 顺序轮流给各相绕组通电，磁场按 A→B→C 反向转过了一周，转子则沿相同方向转子一个步距角。

同样，如果改变通电顺序，按与上面相反的方向（A→C→B→A 顺序）通电，则转子的转向也改变。

如果对绕组通电一次的操作称为一拍，那么前面所述的三相反应式步进电动机的三相轮流通电就需要三拍。转子每拍走一步，转一个齿距角需要 3 步。

转子走一步所转过的角度称为步距角 $\theta_N = 2\pi/Z$，可用下式计算：

$$\theta_N = \theta_Z / N = 2\pi / (NZ)$$

式中，N——步进电动机工作拍数。

例如，对于转子有 40 个齿的三相步进电动机来说，转过一个齿距角相当于转过 9°，共用了 3 步，每换相一次走一步，这样每步走了 3°，步距角为 3°。

从以上分析可知，反应式步进电动机对结构的要求是：

1）定子绕组磁极的分度角（如三相的 120°和 240°）不能被齿距角整除，否则无法形成错齿。

2）定子绕组磁极的分度角被齿距角除后所得的余数（如三相中的 1/3 齿距角和 2/3 齿距角），应是步距角的倍数（1 倍或 2 倍），而且倍数值与相数不能有公因子，否则无法形成对齿。

（2）步进电动机工作方式

三相步进电动机按通电方式分为单三拍工作方式、双三拍工作方式和六拍工作方式。

单三拍工作方式就是按 A→B→C→A 方式循环通电。其中“单”指的是每次对一相通电，所以单三拍工作方式又称为 1 相励磁工作方式；“三拍”指的是磁场旋转一周需要换相 3 次，这时转子转动一个齿距角。

用单三拍工作方式工作时，各相通电的波形如图 11-10 所示。

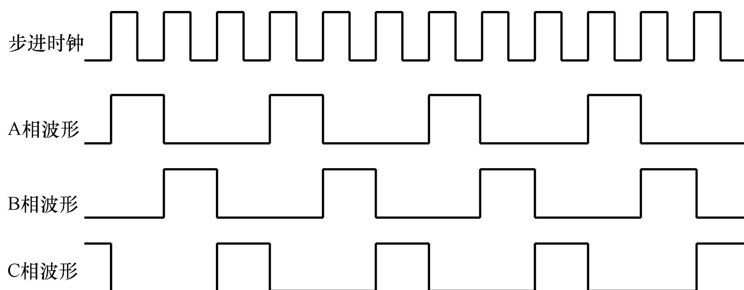


图 11-10　单三拍工作方式波形

单三拍工作方式时，由于只对单相通电，所以产生转矩较小，地磁阻尼也较小，高频性能差，容易产生振荡。

双三拍的工作方式是：每次对两相同时通电，即所谓“双”，所以双三拍工作方式又称为 2 相励磁工作方式；磁场旋转一周需要换相 3 次，即所谓“三拍”，转子转过一个齿距角。在双三拍工作方式中，步进电动机正转的通电顺序为：AB→BC→CA；反转的通电顺序为：BA→AC→CB。

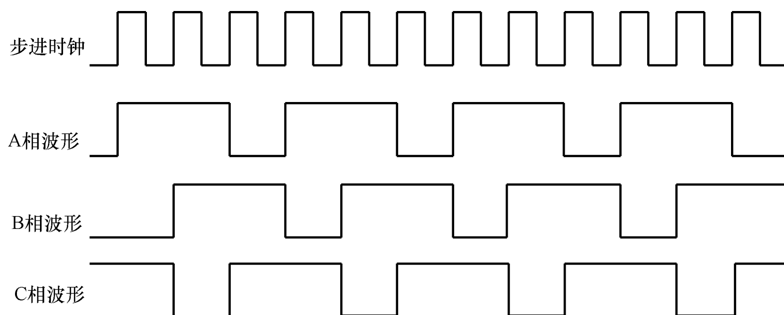


图 11-11　双三拍工作方式波形

在用双三拍方式工作时，各相通电的波形如图 11-11 所示。由图可见，每一拍中，都有两相同时通电，每一相通电时间都持续两拍。注意，双三拍工作时的磁导率最大位置并不是在转子处于对齿的位置。

双三拍工作方式时，因为两相通电后，两相绕组中的电流幅值不同，产生的电磁力方向也不同，所以其中一相产生的电磁力起了阻尼作用。绕组中电流越大，阻尼作用就越大。这有利于步进电动机在低频区工作，不易产生失步。同时，双三拍通电时间长，获得的电磁转矩大，消耗功率也大。

六拍工作方式是单三拍和双三拍交替使用的一种方法，也称作单双六拍或 1-2 相励磁法。步进电动机的正转通电顺序为 A→AB→B→BC→C→CA；反转通电顺序为 A→AC→C→CB→B→BA。可见，磁场旋转一周，通电需要换相 6 次（即六拍），转子才转动一个步距角。

由于转子转动一个步距角需要六拍，所以六拍工作时的步距角要比单三拍和双三拍时的步距角小一半，所以步进精度要高一倍。

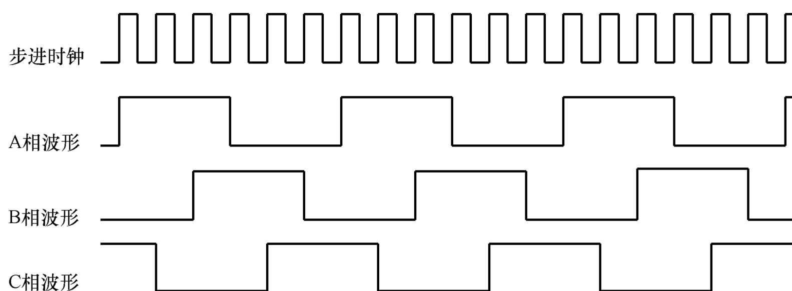


图 11-12　六拍工作方式时的波形图

六拍工作时，各相通电的波形如图 11-12 所示。可以看出，在使用六拍工作方式时，有三拍是单相通电，有三拍是两相通电；对任一相来说，它的电压波形是一个方波，周期为六拍，其中有三拍连续通电，有三拍连续断电。

六拍工作方式除了步进精度高一倍的优点，还能够产生较大转矩，而且功耗适中，高频性能较好。

这三种工作方式的区别较大，一般来说，六拍工作方式的性能最好，单三拍工作方式的性能最差。因此，在步进电动机控制的应用中，选择合适的工作方式非常重要。

6. 步进电动机的失步、振荡及解决方法

步进电动机的失步和振荡是一种普遍存在的现象，它影响了系统的正常运行，因此要尽量避免。下面通过分析失步和振荡的原因，找出较好的解决方法。

（1）失步的原因

1）转子的转速慢于旋转磁场的速度，或者说慢于换相速度。步进电动机在起动时，如果脉冲的频率较高，由于电动机来不及获得足够的能量，使其无法令转子跟上旋转磁场的变化，所以引起失步。

2）转子的平均速度大于旋转磁场的速度。这主要发生在制动和突然换向时，转子获得过多的能量，产生严重的过冲，引起失步。

（2）解决失步的方法

步进电动机有一个技术参数：空载起动频率，即步进电动机在空载情况下能够正常起动

的脉冲频率，如果脉冲频率高于该值，电动机不能正常起动，可能发生丢步或堵转。在有负载的情况下，起动频率应更低。如果要使电动机达到高速转动，脉冲频率应该有加速过程，即起动频率较低，然后按一定加速度升到所希望的高频，电动机转速从低速升到高速。

注意，起动频率不是一个固定值，提高电动机的转矩、减小负载转动惯量、减小步距角都可以提高步进电动机的起动频率。

（3）振荡的原因

步进电动机的振荡现象主要发生于步进电动机工作在低频区、步进电动机工作在共振区和步进电动机突然停车时。

1）当步进电动机工作在低频区时，由于励磁脉冲间隔的时间较长，步进电动机表现为单步运行。当励磁开始时，转子在电磁力的作用下加速转动。在达到平衡点时，电磁驱动转矩为零，但转子的转速最大，由于惯性，转子冲过平衡点。这时电磁力产生负转矩，转子在负转矩的作用下，转速逐渐降为零，并开始反向转动。当转子反转过平衡点后，电磁力又产生正转矩，迫使转子又正向转动。如此循环，形成转子围绕平衡点的振荡。由于有机械摩擦和电磁阻尼的作用，这个振荡变为衰减振荡，最终稳定在平衡点。

2）当步进电动机工作在共振区时，步进电动机的脉冲频率接近步进电动机的振荡频率或振荡频率的分频或倍频，这会使振荡加剧，严重时造成失步。

振荡失步的过程可描述如下：在第 1 个脉冲到来后，转子经历了一次振荡。当转子回摆到最大幅值时，恰好第 2 个脉冲到来，转子受到的电磁转矩为负值，使转子继续回摆。接着第 3 个脉冲到来，转子受正电磁转矩的作用回到平衡点。这样，转子经过 3 个脉冲仍然回到原来位置，也就是丢了 3 步。

3）当步进电动机工作在高频区时，由于换相周期短，转子来不及反冲。同时，绕组中的电流还没有上升到稳定值，转子没有获得足够的能量，所以在这个工作区不会产生振荡。

减小步距角可以减小振荡幅值，以达到削弱振荡的目的。

（4）解决振荡的方法

消除振荡是通过增加阻尼的方法来实现的，主要有机械阻尼法和电子阻尼法。其中机械阻尼法比较单一，就是在电动机轴上加阻尼器。电子阻尼法则有以下几种：

1）多相励磁法。采用多相励磁会产生电磁阻尼，会削弱或消除振荡现象。例如，三相步进电动机的单三拍工作方式改成双三拍工作方式或六拍工作方式。

2）变频变压法。步进电动机在高频和在低频时转子所获得的能量不一样。在低频时，绕组中的电流上升时间长，转子获得能量大，因此容易产生振荡；在高频时则相反。因此，可以设计一种电路，使电压随频率的降低而减小，这样使绕组在低频时的电流减小，可以有效地消除振荡。

3）细分步法。细分步法是将步进电动机绕组中的稳定电流分成若干级，每进一步时，电流升一级。同时，也相对地提高步进频率，使加速过程平稳进行。

4）反相阻尼法。这种方法用于步进电动机的制动。在步进电动机转子要过平衡点之前，加一个反向作用力去平衡惯性力，使转子到达平衡点时的速度为 0，实现准确制动。

11.2.2　步进电动机的控制

步进电动机是纯粹的数字控制电动机。它将电脉冲信号转变成角位移，即给一个脉冲信号，步进电动机就转动一个角度，因此非常适合单片机的控制。

一般一个完整的步进电动机控制系统包括控制器、驱动器、电动机三部分。框图如图 11-13 所示：



图 11-13　步进电动机控制系统框图

本节以反应式步进电动机为例，介绍其基本原理与应用方法。反应式步进电动机可实现大转矩输出，步进角一般为 1.5°。反应式步进电动机的转子磁路由软磁材料制成，定子上有多相励磁绕组，利用磁导的变化产生转矩。常用小型步进电动机的实物如图 11-14 所示。

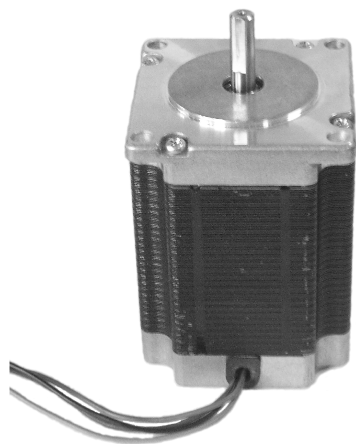


图 11-14　步进电动机实物图

1. 步进电动机的励磁方式

步进电动机的励磁方式一般分为 1 相励磁、2 相励磁、1-2 相励磁。

1 相励磁时，步进电动机按 $A \to B \to \overline{A} \to \overline{B}$ 方式循环通电，每次只对一相通电，磁场旋转一周需要换相 4 次，转子转动一个齿距角。其通电方式最为简单，转矩最小。励磁方式见表 11-1。

表 11-1　1 相励磁方式

步数	A	B	$\overline{A}$	$\overline{B}$
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

2 相励磁时，每次对两相同时通电，磁场旋转一周需要换相 4 次，转子转动一个齿距角。在双三拍工作方式中，步进电动机正转的通电顺序为 $AB \to \overline{A}B \to \overline{A}\overline{B} \to \overline{B}A$；反转的通电顺序为 $BA \to A\overline{B} \to \overline{A}\overline{B} \to B\overline{A}$。双三拍工作方式的优点是，可产生较大的转矩，不易产生失步。励磁方式见表 11-2。

表 11-2　2 相励磁方式

步数	A	B	$\overline{A}$	$\overline{B}$
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

1-2 相励磁是 1 相励磁和 2 相励磁交替使用的方法。磁场旋转一周需要换相 8 次，转子才转过一个步距角，属于半步的方式，也就是说 1-2 相励磁时的步距角比前两种方式的步距角小一半，所以步进精度提高了一倍。1-2 相励磁方式见表 11-3。

2. 步进电动机现场应用驱动电路

在本节中使用的是小型步进电动机，对电压和电流要求不是很高，为了说明应用原理，

表 11-3　1-2 相励磁方式

步数	A	B	$\overline{A}$	$\overline{B}$
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

故采用最简单的驱动电路，目的在于验证步进电动机的使用，在正式工业控制中还需在此基础上改进。一般的驱动电路可以用如图 11-15 所示的形式。

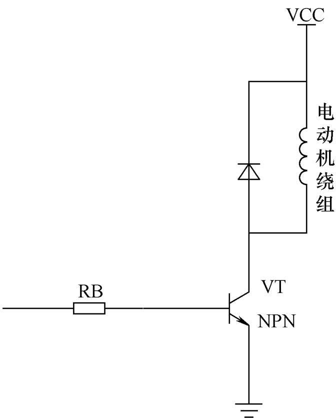


图 11-15　一般驱动电路

在实际应用中一般驱动路数不止一路，用图 11-15 的分立电路体积大，很多场合用现成的集成电路作为多路驱动。常用的小型步进电动机驱动电路可以用 ULN2003 或 ULN2803。本书配套实验板上用的是 ULN2003。ULN2003 是高压大电流达林顿晶体管阵列系列产品，具有电流增益高、工作电压高、温度范围宽、带负载能力强等特点，适应于各类要求高速大功率驱动的系统。ULN2003A 由 7 组达林顿晶体管阵列和相应的电阻网络以及钳位二极管网络构成，具有同时驱动 7 组负载的能力，为单片双极型大功率高速集成电路。ULN2003 内部结构如图 11-16 所示，达林顿晶体管等效电路图如图 11-17 所示。

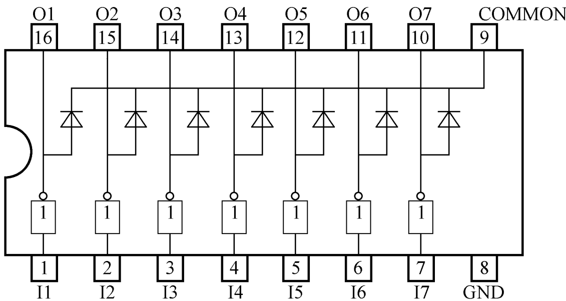


图 11-16　ULN2003 内部结构图

ULN2003A 型高压大电流达林顿晶体管阵列电路的典型应用电路框图如图 11-18 所示。钳位二极管用于保护线圈免于通断时的反电动势击穿集成电路，可以看出，该电路的应用非常简单。

11.2.3　步进电动机的应用设计

1. 硬件原理图设计

电路原理图如图 11-19 所示。OUT1、OUT2、OUT3、OUT4 为电动机脉冲输出引脚，CLAMP 为钳位公共点。

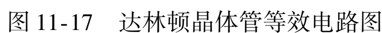


图 11-17　达林顿晶体管等效电路图

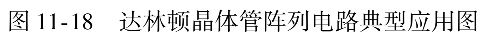


图 11-18　达林顿晶体管阵列电路典型应用图

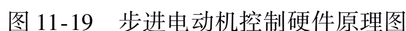


图 11-19　步进电动机控制硬件原理图

2. 软件设计

```
/*********************************************************************/
/*步进电动机测试程序                                                    */
/*目标器件:ATmega16                                                    */
/*晶振:RC 1MHz                                                         */
/*编译环境:ICCAVR 6.31A                                                */
/*********************************************************************/
/**********************包含头文件***********************/
#include <iom16v.h>
#include <macros.h>

/*******************************************************************
函数功能:延时子程序
入口参数:
出口参数:
*******************************************************************/
void delay(void)
{
   unsigned char k;
   for(k=0;k<50;k++);
}

/*******************************************************************
函数功能:主程序
入口参数:
出口参数:
*******************************************************************/
void main(void)
{
      unsigned int i;
      unsigned char j;
      DDRA=0xff;
      PORTA=0xff;
      while(1)
      {//步进电动机正转
            PORTA=0xcf;
            delay();

            PORTA=0x6f;
            delay();
```

```
            PORTA = 0x3f;
            delay( );

            PORTA = 0x9f;
            delay( );
        }
    }
```

11.3　DS18B20 单总线数字温度传感器应用实例

随着现代化信息技术的飞速发展和传统工业改造的逐步实现，能独立工作的温度检测系统已广泛应用于各种不同领域。传统的温度检测系统大多采用热敏电阻作为传感器。采用热敏电阻作为传感器的传统温度检测系统必须经过专门的接口电路转换成数字信号后才能由微处理器进行处理，存在可靠性差、成本高、精度低等诸多缺点，现在很多温度检测控制场合已广泛使用单总线的温度传感器，使整个系统简单可靠。

11.3.1　单总线技术简介

目前，单片机外设的接口形式主要有单总线接口、I^2C 接口、SPI 接口、PS/2 接口等。SPI 接口与单片机通信需要 3 根线，I^2C 接口也要 2 根线，而单总线器件与单片机间数据通信只要 1 根线。美国 DALLAS 公司推出的单总线（1 Wire BUS）技术与 I^2C、SPI、PS/2 总线不同，它采用单根信号线，既可以传输时钟信号又可以传送数据信号，而数据又可双向传送，因而这种总线技术具有线路简单、成本低廉、便于扩展和维护等优点。

单总线适用于单主机系统，能够控制一个或多个从机设备。主机可以是微处理器，从机可以是单总线器件，它们之间的数据交换只通过一条信号线。当只有一个从机设备时，系统可按单节点系统操作；当有多个从设备时，系统则按多节点系统操作。主机或从机通过一个漏极开路或三态端口连接到这个数据线，以允许设备在不发送数据时能够释放总线，而让其他设备使用总线，其内部等效电路图如图 11-20 所示。单总线通常要求接一个约为 4.7kΩ 的上拉电阻，这样，当总线空闲时，其状态为高电平。主机和从机之间的通信可以通过 3 个

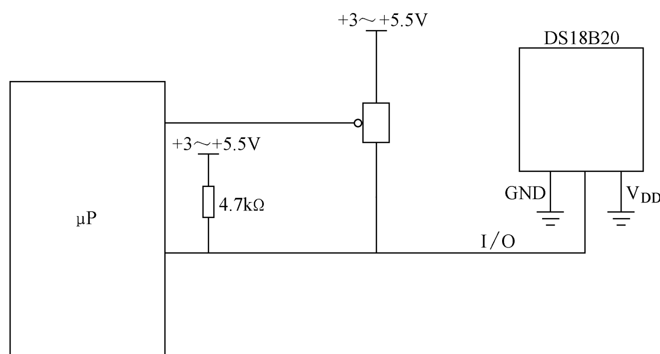


图 11-20　内部等效电路图

步骤完成，分别是初始化单总线器件、识别单总线器件、数据交换。由于它们是主从结构，只有主机呼号从机时，从机才能应答，因此主机访问单总线器件时都必须严格遵循单总线命令序列。如果出现序列混乱，单总线器件将不响应主机。

所有的单总线器件都要遵循严格的通信协议，以保证数据的完整性。单总线协议定义了复位信号、应答信号、写“0”、读“0”、写“1”、读“1”的几种时序信号类型。所有的单总线命令序列都是由这些基本的信号类型组成的。在这些信号中，除了应答脉冲外，其他均由主机发出同步信号，并且发送的所有命令和数据都是字节的低位在前。

所有单总线器件的读、写时序至少需要 60μs，且每两个独立的时序间至少需要 1μs 的恢复时间。在写时序中，主机将在拉低总线 15μs 之内释放总线，并向单总线器件写“1”；如果主机拉低总线后能保持至少 60μs 的低电平，则向总线器件写“0”。单总线器件仅在主机发出读时序时才向主机传输数据，所以，当主机向单总线器件发出读数据命令后，必须马上产生读时序，以便单总线器件能传输数据。

11.3.2　DS18B20 单总线温度传感器简介

在多点温度测量系统中，单总线数字温度传感器因其体积小、构成的系统结构简单等优点，应用越来越广泛。每一个数字温度传感器内均有唯一的 64 位序列号（最低 8 位是产品代码，其后 48 位是器件序列号，最后 8 位是前 56 位循环冗余校验码），只有获得该序列号后才可能对其进行操作，也才能在多传感器系统中将它们一一识别。

DS18B20 是 DALLAS 公司生产的单总线式数字温度传感器，它具有微型化、低功耗、高性能、抗干扰能力强、易配处理器等优点，特别适用于构成多点温度测控系统，可直接将温度转化成串行数字信号（提供 9 位二进制数字）给单片机处理，且在同一总线上可以挂接多个传感器芯片。它具有 3 引脚 TO－92 小体积封装形式，温度测量范围为－55～＋125℃，可编程为 9～12 位 A/D 转换精度，测温分辨率可达 0.0625℃，被测温度用符号扩展的 16 位数字量方式串行输出，其工作电源既可在远端引入，也可采用寄生电源方式产生，多个 DS18B20 可以并联到 3 根或 2 根线上，CPU 只需 1 根端口线就能与多个 DS18B20 通信，占用微处理器的端口较少，可节省大量的引线和逻辑电路。以上特点使 DS18B20 非常适用于远距离多点温度检测系统。DS18B20 还具有以下新特性：

1）独特的单线接口仅需一个端口引脚进行通信。

2）简单的多点分布应用。

3）无需外部器件。

4）可通过数据线供电。

5）零待机功耗。

6）测温范围－55～＋125℃，以 0.5℃递增。华氏器件－67～＋2570°F，以 0.90°F 递增。

7）可编程的分辨率为 9～12 位，对应的可分辨温度分别为 0.5℃、0.25℃、0.125℃和 0.0625℃。

8）温度数字量转换时间为 200ms（典型值），12 位分辨率时最多在 750ms 内把温度值转换为数字。

9）用户可定义的非易失性温度报警设置。

10）报警搜索命令识别并标志超过程序限定温度（温度报警条件）的器件。

11）应用包括温度控制、工业系统、消费品、温度计或任何热感测系统。

12）负压特性：电源极性接反时，温度计不会因发热而烧毁，但不能正常工作。

1. DS18B20 外形及引脚说明

外形及引脚如图 11-21 所示。

在 TO-92 和 SO-8 的封装中引脚有所不同，具体差别请查阅相关手册，在 TO-92 封装中引脚分配如下：

1（GND）：地。

2（DQ）：单线运用的数据输入输出引脚。

3（VDD）：可选的电源引脚。

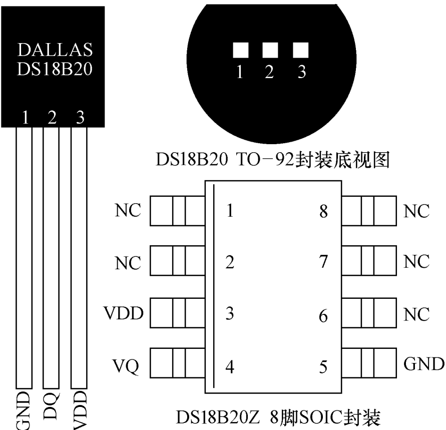


图 11-21　外形及引脚

2. DS18B20 的内部结构

DS18B20 内部结构如图 11-22 所示，主要由 4 部分组成：64 位 ROM、温度传感器、非挥发的温度报警触发器 TH 和 TL、配置寄存器。ROM 中的 64 位序列号是出厂前被光刻好的，它可以看作是该 DS18B20 的地址序列码，每个 DS18B20 的 64 位序列号均不相同。64 位激光 ROM 从高位到低位依次为 8 位 CRC、48 位序列号和 8 位家族代码（28H）组成。ROM 的作用是使每一个 DS18B20 都各不相同，这样就可以实现一根总线上挂接多个 DS18B20 的目的。非易失性温度报警触发器 TH 和 TL 可通过软件写入用户报警上下限值。配置寄存器为高速暂存存储器中的第 5 个字节。DS18B20 在工作时按此寄存器中的分辨率将温度转换成相应精度的数值，其各位定义如图 11-23 所示。其中，TM：测试模式标志位，出厂时被写入 0，不能改变；

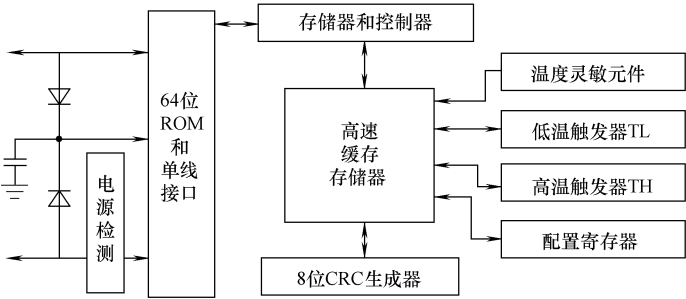


图 11-22　内部结构

TM	R1	R0	1	1	1	1	1
MSB							LSB

图 11-23　寄存器各位定义图

R0	R1	温度计分辨率/bit	最大转换时间/ms
0	0	9	93.75
0	1	10	187.5
1	0	11	375
1	1	12	750

图 11-24　配置寄存器与分辨率关系表

R0、R1：温度计分辨率设置位，其对应 4 种分辨率如图 11-24 所示。出厂时 R0、R1 置为默认值：R0 = 1，R1 = 1（即 12 位分辨率），用户可根据需要改写配置寄存器，以获得合适的分辨率。

3. DS18B20 工作过程及时序

DS18B20 内部的低温度系数振荡器是一个振荡频率随温度变化很小的振荡器，为计数器 1 提供频率稳定的计数脉冲。

高温度系数振荡器是一个振荡频率对温度很敏感的振荡器，为计数器 2 提供频率随温度变化的计数脉冲。

初始时，温度寄存器被预置成 –55℃，每当计数器 1 从预置数开始减计数到 0 时，温度寄存器中寄存的温度值就增加 1℃，这个过程重复进行，直到计数器 2 计数到 0 时便停止。

初始时，计数器 1 预置的是与 –55℃相对应的一个预置值。以后计数器 1 每一个循环的预置数都由斜率累加器提供。为了补偿振荡器温度特性的非线性特性，斜率累加器提供的预置数也随温度相应变化。计数器 1 的预置数也就是在给定温度处使温度寄存器寄存值增加 1℃计数器所需要的计数个数。

DS18B20 内部的比较器以四舍五入的量化方式确定温度寄存器的最低有效位。在计数器 2 停止计数后，比较器将计数器 1 中的计数剩余值转换为温度值后与 0. 25℃进行比较，若低于 0. 25℃，温度寄存器的最低位就置 0；若高于 0. 25℃，最低位就置 1；若高于 0. 75℃时，温度寄存器的最低位就进位然后置 0。这样，经过比较后所得的温度寄存器的值就是最终读取的温度值了，其最后位代表 0. 5℃，四舍五入最大量化误差为 ±1/2LSB，即 0. 25℃。

温度寄存器中的温度值以 9 位数据格式表示，最高位为符号位，其余 8 位以二进制补码形式表示温度值。测温结束时，这 9 位数据转存到暂存存储器的前两个字节中，符号位占用第 1 字节，8 位温度数据占据第 2 字节。

DS18B20 测量温度时使用特有的温度测量技术。DS18B20 内部的低温度系数振荡器能产生稳定的频率信号；同样的，高温度系数振荡器则将被测温度转换成频率信号。当计数门打开时，DS18B20 进行计数，计数门开通时间由高温度系数振荡器决定。芯片内部还有斜率累加器，可对频率的非线性度加以补偿。测量结果存入温度寄存器中。一般情况下的温度值应该为 9 位，但因符号位扩展成高 8 位，所以最后以 16 位补码形式读出。

DS18B20 工作过程一般遵循以下协议：初始化——ROM 操作命令——存储器操作命令——处理数据。

（1）初始化

单总线上的所有处理均从初始化序列开始。初始化序列包括总线主机发出一复位脉冲，接着由从属器件送出存在脉冲。存在脉冲让总线控制器知道 DS18B20 在总线上且已准备好操作。

（2）ROM 操作命令

一旦总线主机检测到从属器件的存在，它便可以发出器件 ROM 操作命令之一。所有 ROM 操作命令均为 8 位。这些命令列表如下：

Read ROM（读 ROM）[33h]

此命令允许总线主机读 DS18B20 的 8 位产品系列编码、唯一的 48 位序列号，以及 8 位

的 CRC。此命令只能在总线上仅有一个 DS18B20 的情况下可以使用。如果总线上存在多于一个的从属器件，那么当所有从片企图同时发送时将发生数据冲突的现象（漏极开路会产生线与的结果）。

Match ROM（符合 ROM）[55h]

此命令后继以 64 位的 ROM 数据序列，允许总线主机对多点总线上特定的 DS18B20 寻址。只有与 64 位 ROM 序列严格相符的 DS18B20 才能对后继的存储器操作命令作出响应。所有与 64 位 ROM 序列不符的从片将等待复位脉冲。此命令在总线上有单个或多个器件的情况下均可使用。

Skip ROM（跳过 ROM ）[CCh]

在单点总线系统中，此命令通过允许总线主机不提供 64 位 ROM 编码而访问存储器操作来节省时间。如果在总线上存在多于一个的从属器件，而且在 Skip ROM 命令之后发出读命令，那么由于多个从片同时发送数据，会在总线上发生数据冲突（漏极开路下拉会产生线与的效果）。

Search ROM（搜索 ROM）[F0h]

当系统开始工作时，总线主机可能不知道单线总线上的器件个数或者不知道其 64 位 ROM 编码。搜索 ROM 命令允许总线控制器用排除法识别总线上所有从机的 64 位编码。

Alarm Search（告警搜索）[ECh]

此命令的流程与搜索 ROM 命令相同。但是，仅在最近一次温度测量出现告警的情况下，DS18B20 才对此命令作出响应。告警的条件定义为温度高于 TH 或低于 TL。只要 DS18B20 一上电，告警条件就保持在设置状态，直到另一次温度测量显示出非告警值或者改变 TH 或 TL 的设置，使得测量值再一次位于允许的范围之内。储存在 EEPROM 内的触发器值用于告警。

（3）存储器操作命令

Write Scratchpad（写暂存存储器）[4Eh]

这个命令向 DS18B20 的暂存器中写入数据，开始位置在地址 2。接下来写入的两个字节将被存到暂存器中的地址位置 2 和 3。可以在任何时刻发出复位命令来中止写入。

Read Scratchpad（读暂存存储器）[BEh]

这个命令读取暂存器的内容。读取将从字节 0 开始，一直进行下去，直到第 9（字节 8，CRC）字节读完。如果不想读完所有字节，控制器可以在任何时间发出复位命令来中止读取。

Copy Scratchpad（复制暂存存储器）[48h]

这条命令把暂存器的内容复制到 DS18B20 的 EEPROM 里，即把温度报警触发字节存入非易失性存储器里。如果总线控制器在这条命令之后跟着发出读时间隙，而 DS18B20 又正在忙于把暂存器复制到 EEPROM，DS18B20 就会输出一个“0”，如果复制结束的话，DS18B20 则输出“1”。如果使用寄生电源，总线控制器必须在这条命令发出后立即起动强上拉并最少保持 10ms。

Convert T（温度变换）[44h]

这条命令起动一次温度转换而无需其他数据。温度转换命令被执行，而后 DS18B20 保持等待状态。如果总线控制器在这条命令之后跟着发出读时间隙，而 DS18B20 又忙于做时

间转换的话，DS18B20 将在总线上输出“0”，若温度转换完成，则输出“1”。如果使用寄生电源，总线控制器必须在发出这条命令后立即起动强上拉，并保持 500ms。

Recall E^2（重新调整 EEPROM）［B8h］

这条命令把储存在 EEPROM 中温度触发器的值重新调至暂存存储器。这种重新调出的操作在对 DS18B20 上电时也自动发生，因此只要器件一上电，暂存存储器内就有了有效的数据。在这条命令发出之后，对于所发出的第一个读数据时间片，器件会输出温度转换忙的标识“0”=忙，“1”=准备就绪。

Read Power Supply（读电源）［B4h］

对于在此命令发送至 DS18B20 之后所发出的第一个读数据的时间片，器件都会给出其电源方式的信号“0”=寄生电源供电，“1”=外部电源供电。

（4）处理数据

DS18B20 的高速暂存存储器由 9 个字节组成，其分配如图 11-25 所示。当温度转换命令发布后，经转换所得的温度值以二进制补码形式存放在高速暂存存储器的第 0 和第 1 个字节。单片机可通过单线接口读到该数据，读取时低位在前，高位在后。

温度低位	温度高位	TH	TL	配置	保留	保留	保留	8位CRC
LSB								MSB

图 11-25　高速暂存存储器分配图

表 11-4　DS18B20 温度数据表

温度/℃	二进制表示												十六进制表示
	符号位(5 位)	数据位(11 位)											
+125	0 0 0 0 0	1	1	1	1	1	0	1	0	0	0	0	07D0H
+25.0625	0 0 0 0 0	0	0	1	1	0	0	1	0	0	0	1	0191H
+10.125	0 0 0 0 0	0	0	0	1	0	1	0	0	0	1	0	00A2H
+0.5	0 0 0 0 0	0	0	0	0	0	0	0	1	0	0	0	0008H
0	0 0 0 0 0	0	0	0	0	0	0	0	0	0	0	0	0000H
−0.5	1 1 1 1 1	1	1	1	1	1	1	1	1	0	0	0	FFF8H
−10.125	1 1 1 1 1	1	1	1	0	1	0	1	1	1	1	0	FF5EH
−25.625	1 1 1 1 1	1	1	0	0	1	1	0	1	1	1	1	FE6FH
−55	1 1 1 1 1	1	0	0	1	0	0	1	0	0	0	0	FC90H

表 11-4 是 DS18B20 温度采集转化后得到的 16 位数据，存储在 DS18B20 的两个 8bit 的 RAM 中，二进制中的前面 5 位是符号位，如果测得的温度大于或等于 0，这 5 位为 0，只要将测到的数值乘以 0.0625 即可得到实际温度；如果温度小于 0，这 5 位均为 1，测到的数值需要取反加 1 再乘以 0.0625 即可得到实际温度。

温度转换计算方法举例：

例如，当 DS18B20 采集到 +125℃的实际温度后，输出为 07D0H，则

实际温度 = 07D0H × 0.0625℃ = 2000 × 0.0625℃ = 125℃。

例如，当 DS18B20 采集到 −55℃的实际温度后，输出为 FC90H，则应先将 11 位数据位

取反加 1 得 370H（符号位不变，也不作为计算），则

实际温度 = 370H × 0.0625℃ = 880 × 0.0625℃ = 55℃。

（5）时序

主机使用时间隙来读写 DS18B20 的数据位和写命令字的位。初始化及读写时序如图 11-26和图 11-27 所示。

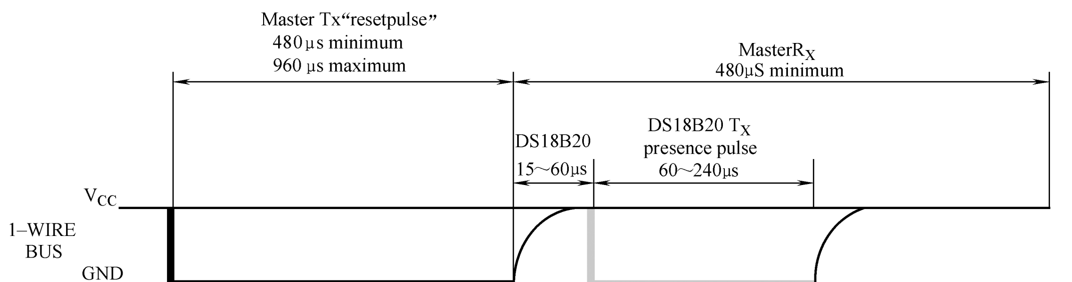


图 11-26　初始化时序

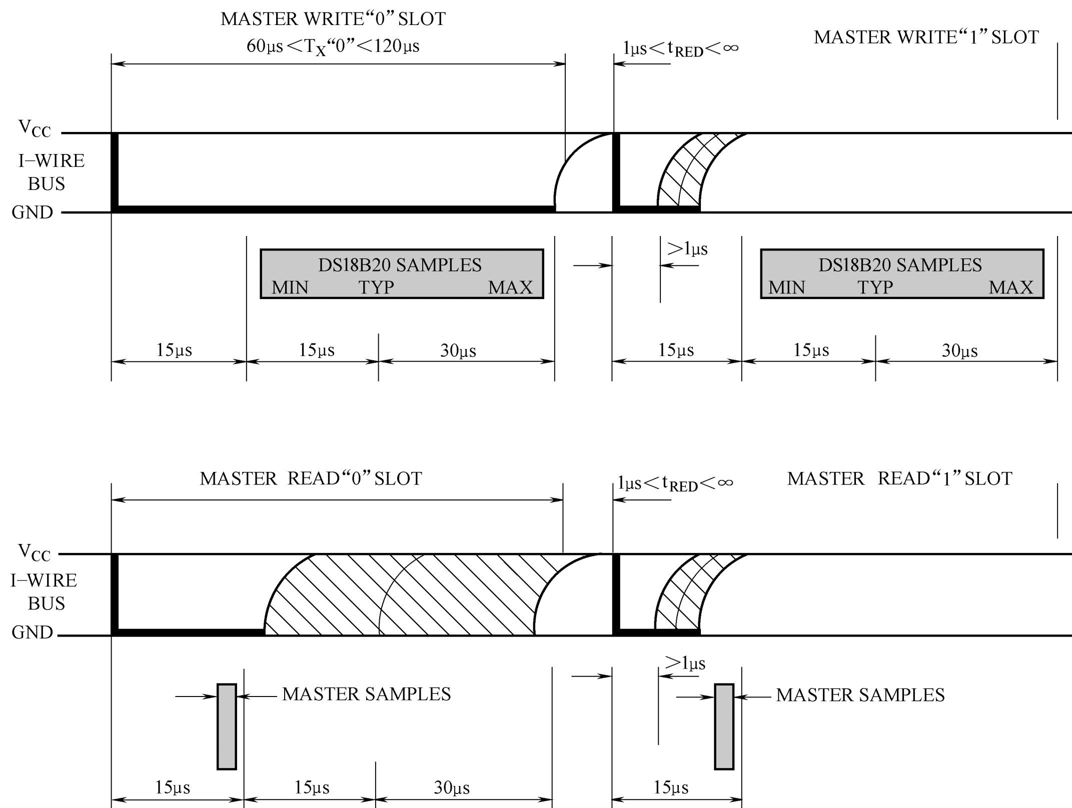


图 11-27　读写时序

11.3.3　DS18B20 软硬件设计

本实例介绍 DS18B20 与单片机之间的软、硬件接口，通过单片机来读取 DS18B20 的温度值，并将温度值通过数码管显示出来。

1. 硬件原理图（见图 11-28）

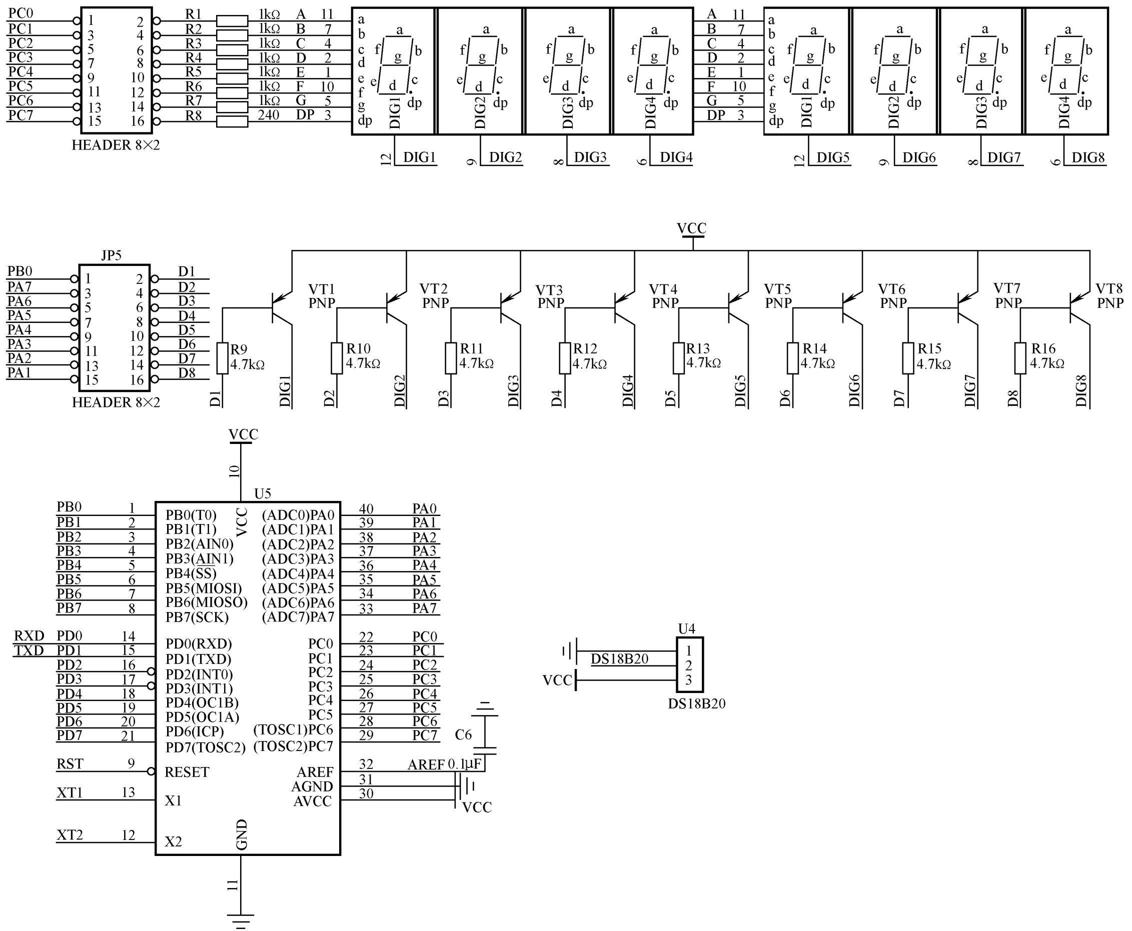


图 11-28　硬件原理图

2. 软件设计

```
/*****************************************************************/
/* DS18B20 测试程序                                               */
/* 目标器件:ATmega16                                              */
/* 晶振:RC 1MHz                                                   */
/* 编译环境:ICCAVR 6.31A                                          */
/*****************************************************************/
/********************包含头文件****************************/
#include <iom16v.h>
#include <macros.h>


/*********************数码管段码表*************************/
extern const unsigned char tab[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,
```

```
0x90,
                  /*    0    1    2    3    4    5    6    7    8    9   */
                              0x88,0x83,0xC6,0xA1,0x86,0x8E};
                  /*    a    b    c  d    e    f   */

/*************************端口定义*************************/
#define    DQon          PORTB |=(1<<2);
#define    DQoff      PORTB &= ~(1<<2);

/*************************定义全局变量*************************/
unsigned char tempL=0;              //临时变量低位
unsigned char tempH=0;              //临时变量高位
float temperature;                  //温度值
float T;

/***********************************************************
函数功能:延时子程序
入口参数:k
出口参数:
***********************************************************/
void delay(unsigned int k)
{
    unsigned int n;
    n=0;
    while(n< k)
    {n++;}
    return;
}

/***********************************************************
函数功能:数码管延时程序
入口参数:
出口参数:
***********************************************************/
void delay_1us(void)                //1μs 延时函数
{
    asm("nop");
}

void delay_nus(unsigned int n)      //Nμs 延时函数
```

```
{
    unsigned int i = 0;
    for(i = 0;i < n;i ++)
    delay_1us();
}

void delay_1ms(void)                //1ms 延时函数
{
    unsigned int i;
    for(i = 0;i < 140;i ++);
}

void delay_nms(unsigned int n)      //N ms 延时函数
{
    unsigned int i = 0;
    for(i = 0;i < n;i ++)
    delay_1ms();
}

/*************************************************************
函数功能:显示子程序
入口参数:k
出口参数:
*************************************************************/
void display(unsigned int k)
{
    PORTC = tab[k/1000];
    PORTA = 0xef;
    delay_nms(2);

    PORTC = tab[k/100%10];
    PORTA = 0xf7;
    delay_nms(2);

    PORTC = tab[k/10%10];
    PORTA = 0xfb;
    delay_nms(2);

    PORTC = tab[k%10];
    PORTA = 0xfd;
```

```
    delay_nms(2);
}


/**********************************************************
函数功能:DS18B20 初始化子程序
入口参数:
出口参数:
**********************************************************/
Init_DS18B20(void)
{
    unsigned char x=0;
    DQon;                              //DQ 先置高
    delay(8);                          //延时
    DQoff;                             //发送复位脉冲
    delay(85);                         //延时(>480ms)
    DQon;                              //拉高数据线
    delay(14);                         //等待(15~60ms)
}

/**********************************************************
函数功能:向 DS18B20 读一个字节数据
入口参数:
出口参数:dat
**********************************************************/
ReadOneChar(void)
{
    unsigned char i=0;
    unsigned char dat=0;
    for(i=8;i>0;i--)
    {
      DQon;
      delay(1);
      DQoff;
      dat>>=1;
      DQon;              //input
      DDRB=0xfb;
      if(PINB&0x04)
        dat|=0x80;
      DDRB=0xff;
```

```
        delay(4);
    }
    return(dat);
}

/*************************************************************
函数功能:向 DS18B20 写一个字节数据
入口参数:dat
出口参数:
*************************************************************/
WriteOneChar(unsigned char dat)
{
    unsigned char i =0;
    for(i =8;i >0;i - - )
    {
        DQoff;
        if(dat&0x01)
        {DQon;}
        else
        {DQoff;}
        delay(5);
        DQon;
        dat > > =1;
    }
    delay(4);
}

/*************************************************************
函数功能:向 DS18B20 读温度值
入口参数:
出口参数:temperature
*************************************************************/
ReadTemperature(void)
{
    Init_DS18B20();                 //初始化
    WriteOneChar(0xcc);             //跳过读序列号的操作
    WriteOneChar(0x44);             //起动温度转换
    delay(125);                     //转换需要一点时间,延时
    Init_DS18B20();                 //初始化
    WriteOneChar(0xcc);             //跳过读序列号的操作
```

```
        WriteOneChar(0xbe);                 //读温度寄存器(前两个值分别为温度的低位和
高位)
        tempL = ReadOneChar();              //读出温度的低位 LSB
        tempH = ReadOneChar();              //读出温度的高位 MSB
        //温度转换,把高低位做相应的运算转化为实际温度
        temperature = ((tempH * 256) + tempL) * 0.0625;
        delay(200);
        return(temperature);
    }

    /*************************************************************
    函数功能:主程序
    入口参数:
    出口参数:
    *************************************************************/
    void main(void)
    {
        DDRC = 0xff;
        PORTC = 0xff;
        DDRA = 0xfe;
        PORTA = 0xfe;
        DDRB = 0xfe;
        PORTB = 0xff;

        display(0);
        while(1)
        {
        T = ReadTemperature();
        display(T);
        }
        NOP();
    }
```

11.4　I^2C 总线应用实例

I^2C（Inter Integrated Circuit）总线实际上已经成为一个国际标准，已在超过 100 种不同的 IC 上实现，而且得到超过 50 家公司的许可。I^2C 总线通过一个简单的双向两线总线实现了有效的 IC 之间控制，使厂商和设计者都从中得益，减少了软硬件的开发时间。

AVR 单片机内部集成了 TWI（Two Wire Serial Interface）串行总线接口。该接口是一个面向字节和基于中断的硬件接口，弥补了某些单片机只能依靠时序模拟完成 I^2C 总线工作的

缺陷，在操作和使用上比 I^2C 总线更为灵活。

本节将在学习 I^2C 总线协议的基础上，介绍 AVR 单片机的 TWI 的特点，并以读/写 24C 系列存储器为例，说明 I^2C 总线在实际中的应用。

11.4.1　I^2C 串行总线简介

I^2C 总线是 Philips 公司推出的一种用于器件之间连接的 2 线制串行总线，是具备多主机系统所需的包括总线裁决和高低速器件同步功能的高性能串行总线。表 11-5 为 I^2C 总线术语解释。

表 11-5　I^2C 总线术语解释

术　语	描　述
发送器	发送数据到总线的器件
接收器	从总线接收数据的器件
主机	初始化发送产生时钟信号和终止发送的器件
从机	被主机寻址的器件
多主机	同时有多于一个主机尝试控制总线但不破坏报文
仲裁	一个在有多个主机同时尝试控制总线但只允许其中一个控制总线并使报文不被破坏的过程
同步	两个或多个器件同步时钟信号的过程

I^2C 总线只有两根双向信号线，一根是数据线 SDA，令一根是时钟线 SCL。所有连接到 I^2C 总线上器件的数据线都连接到 SDA 线上，各器件的时钟线均接到 SCL 线上。I^2C 总线的基本结构如图 11-29 所示。

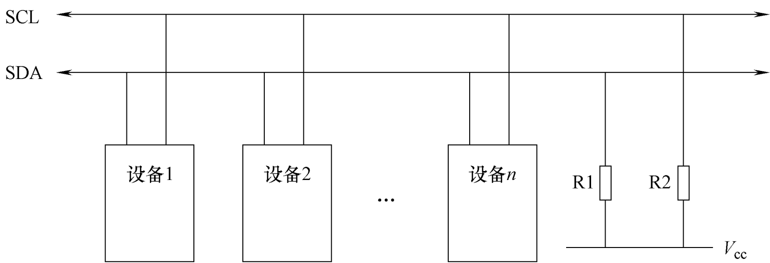


图 11-29　I^2C 总线基本结构

I^2C 总线是一个多主机总线，即总线上可以有一个或多个主机，总线运行由主机控制。通常，主机由各种单片机或其他微处理器充当。从机可以是各种单片机或其他处理器，也可以是其他器件，如存储器、LCD 或 LED 驱动器、A/D 或 D/A 转换器、存储器或键盘接口等。每个接到 I^2C 总线上的器件都有一个唯一的地址识别，而且都可以作为一个发送器或接收器（由器件的功能决定）。

I^2C 总线的 SDA 和 SCL 是双向的，均通过上拉电阻接正电源，如图 11-29 所示。当总线空闲时，两根线均为高电平。连接到总线上的器件的输出级必须是漏极或集电极开路的，任一器件输出的低电平，都将使总线的信号变低，即各器件的 SDA 和 SCL 都是线“与”关系。

注意，与总线连接的器件数目受如下条件限制，总线上连接的器件越多，电容值越大，总线上的连接器件的总电容量要低于 400pF。在 I^2C 标准模式下，总线速度可达 100kbit/s，

快速模式下可达 400kbit/s。

下面分析一下数据在两个连接到 I^2C 总线的微控制器之间传输的过程。

传输的过程体现了 I^2C 总线的主机—从机和接收器—发送器的关系。应当注意的是，这些关系不是持久不变的，只由当时数据传输的方向决定。传输数据的过程如下：

（1）假设微控制器 A 要发送信息到微控制器 B

1）微控制器 A（主机）寻址微控制器 B（从机）。

2）微控制器 A（主机—发送器）发送数据到微控制器 B（从机—接收器）。

3）微控制器 A 终止传输。

（2）如果微控制器 A 从微控制器 B 接收信息

1）微控制器 A（主机）寻址微控制器 B（从机）。

2）微控制器 A（主机—接收器）从微控制器 B（从机）发送器接收数据。

3）微控制器 A 终止传输

连接多于一个微控制器到 I^2C 总线的可能性意味着超过一个主机可以同时尝试初始化传输数据。为了避免由此产生混乱，发展出一个仲裁过程。

首先，将所有的主机时钟进行与操作，会生成组合的时钟，其高电平时间等于所有主机中高电平时间最短的一个；低电平时间则等于所有主机中低电平时间最长的一个。

然后，总线仲裁的过程是：如果两个或多个主机尝试发送信息到总线，在其他主机都产生“0”的情况下，首先产生一个“1”的主机将丢失仲裁。仲裁时的时钟信号是用“线与”连接到 SCL 线的主机产生的时钟的同步结合。由于它依靠“线与”连接所有 I^2C 总线接口到 I^2C 总线，某个主机发出的“1”会被其他主机发出的“0”所屏蔽。仲裁是在起始信号后的第一位开始，并逐位进行。由于 SDA 线上的仲裁结果产生的数据在 SCL 为高电平期间总是与掌握控制权的主机发出的数据相同，所以在整个仲裁过程中，SDA 线上的数据完全和最终取得总线控制权的主机发出数据相同。

11.4.2　I^2C 总线器件工作原理及时序

1. 数据的有效性

I^2C 总线进行数据传输时，SDA 线上的数据必须在时钟的高电平周期保持稳定。数据线的高或低电平状态只有在 SCL 线的时钟信号是低电平时才能改变，如图 11-30 所示。

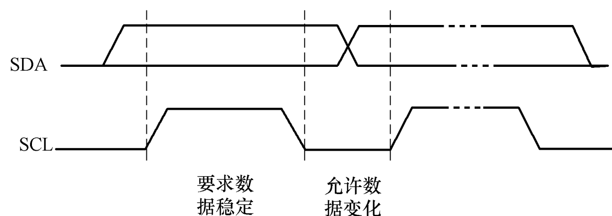


图 11-30　数据位的有效性规定

2. 起始和停止条件

根据 I^2C 总线协议的规定，在 SCL 线是高电平时，SDA 线从高电平向低电平切换表示起始条件；当 SCL 线是高电平时，SDA 线由低电平向高电平切换表示停止条件，如图 11-31 所示。

起始和停止条件一般由主机产生。总线在起始条件后被认为处于忙的状态。在停止条件

的某段时间后总线被认为再次处于空闲状态。

如果连接到总线的器件合并了必要的接口硬件，那么用它们检测起始和停止条件十分简便。但是，没有这种接口的微控制器在每个时钟周期至少要采样 SDA 线两次来判别有没有发生电平切换。

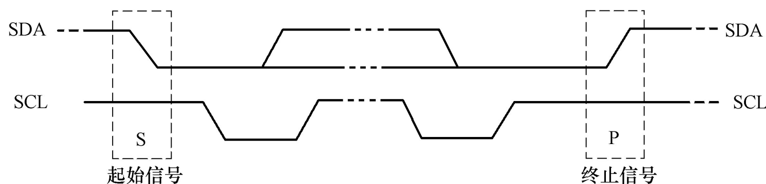


图 11-31　起始和停止信号

3. 传输数据

（1）字节格式与应答

发送到 SDA 线上的每个字节必须为 8 位。每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位（ACK）。首先传输的是数据的最高位（MSB），如图 11-32 所示。如果从机要完成一些其他功能后（例如一个内部中断服务程序）才能接收或发送下一个完整的数据字节，可以使时钟线 SCL 保持低电平迫使主机进入等待状态。当从机准备好接收下一个数据字节并释放时钟线 SCL 后，数据传输继续。

数据传输必须带响应。相关的响应时钟脉冲由主机产生。在响应的时钟脉冲期间，发送器释放 SDA 线（高电平）。在响应的时钟脉冲期间，接收器必须将 SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平。

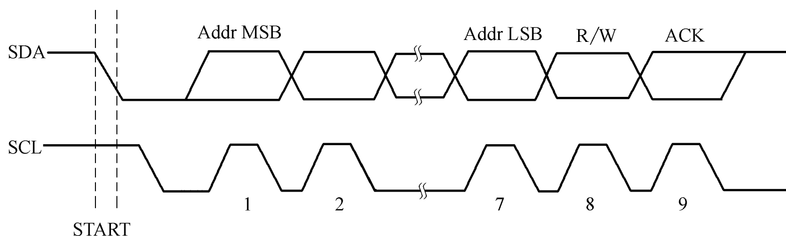


图 11-32　字节格式与应答

当从机不能响应从机地址时（例如，它正在执行一些实时函数不能接收或发送），从机必须使数据线保持高电平。主机然后产生一个停止条件终止传输或者产生重复起始条件开始新的传输。如果从机—接收器响应了从机地址，但是在传输了一段时间后不能接收更多数据字节，主机必须再一次终止。传输这个情况用从机在第 1 个字节后没有产生响应来表示。从机使数据线保持高电平，主机产生一个停止或重复起始条件。

当主机接收数据时，它收到最后一个数据字节后，必须向从机发出一个结束传送的信号。这个信号是由主机对从机的“非应答”来实现的。然后，从机释放 SDA 线，以允许主机产生终止或重复起始信号。

（2）数据帧格式

I^2C 总线上传输的数据信号是广义的，既包括地址信号，又包括真正的数据信号。

I^2C 总线规定，在起始条件（S）后发送了一个从机地址，这个地址共有 7 位，紧接着的第 8 位是数据方向位（R/$\overline{W}$），用“0”表示发送数据（$\overline{W}$），“1”表示接收数据（R）。

数据传输一般由主机产生的停止位（P）终止，如图 11-33 所示。但是如果主机仍希望在总线上通信，它可以产生重复起始条件（Sr）和寻址另一个从机，而不是首先产生一个停止条件。

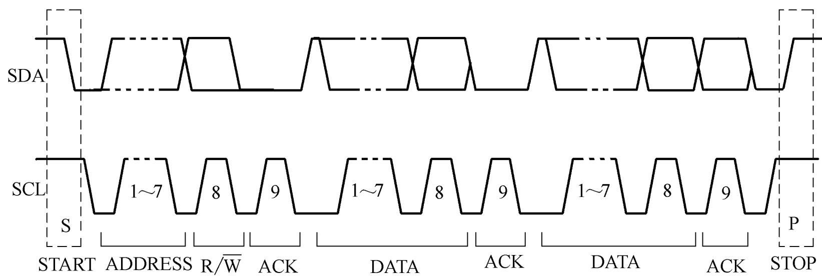


图 11-33　数据传输格式

在这种传输中，可以有以下几种读/写的组合方式：

1）主机向从机发送数据，数据的传送方向在传输过程中不改变，如图 11-34 所示。

S	从机地址	0	A	数据	A	数据	A/$\overline{A}$	P

图 11-34　主机向从机发送数据

注：阴影部分：表示主机向从机发送数据；无阴影部分：表示主机向从机读取数据。
A：表示应答；$\overline{A}$：表示非应答。S：起始信号；P：终止信号。

2）主机在第 1 个字节后，立即向从机读取数据，如图 11-35 所示。

S	从机地址	1	A	数据	A	数据	$\overline{A}$	P

图 11-35　主机在第 1 个字节后立即读从机

3）复合格式，如图 11-36 所示。传输改变方向时，起始条件和从机地址都会被重复，但 R/$\overline{W}$位取反。如果主机接收器发送一个停止或重复起始信号，它之前应该发送了一个不响应信号（$\overline{A}$）。

S	从机地址	0	A	数据	A/$\overline{A}$	S	从机地址	1	A	数据	$\overline{A}$	P

图 11-36　复合格式

由以上格式可见，无论哪种传输方式，起始信号、终止信号和地址均由主机发出（图中阴影部分），数据字节的传送方向则由寻址字节中的方向位规定，每个字节的传送都必须有应答位（A 或$\overline{A}$）。

4. I^2C 总线的寻址

（1）第 1 个字节的位定义

I^2C 总线的寻址过程是通常在起始条件后的第 1 个字节决定了主机选择哪一个从机。

第 1 个字节的前 7 位组成了从机地址，如图 11-37 所示。最低位（LSB）是第 8 位。它

MSB（7）	6	5	4	3	2	1　LSB	（0）
从机地址							R/$\overline{W}$

图 11-37　寻址字节的格式

决定了报文的方向。第一个字节的最低位是“0”表示主机会写数据到被选中的从机，“1”表示主机会向从机读数据。

当发送了一个地址后，系统中的每个器件都将起始条件后的前 7 位与它自己的地址比较。如果一样，器件则认为自己被主机寻址，至于是从机接收器还是从机发送器都由 $R/\overline{W}$ 位决定。

从机地址由一个固定和一个可编程的部分构成。由于很可能在一个系统中有几个同样的器件，从机地址的可编程部分决定了这些器件可以连接到 I^2C 总线上的最大数量。器件可编程地址位的数量由它可使用的引脚决定。例如，如果器件有 4 个固定的和 3 个可编程的地址位，那么相同的总线上共可以连接 8 个相同的器件。

（2）寻址字节中的特殊地址

I^2C 总线规定了一些特殊地址。两组 8 位地址（0000XXXX 和 1111XXXX）已被保留作为特殊用途，见表 11-6。

表 11-6　I^2C 总线特殊地址表

从 机 地 址	$R/\overline{W}$	描　　述
0000 000	0	通用呼叫地址
0000 000	1	起始地址
0000 001	X	CBUS 地址
0000 010	X	保留给不同总线的地址
0000 011	X	
0000 1XX	X	保留
1111 1XX	X	
1111 0XX	X	十位从机地址

起始信号后的第 1 个字节的 8 位为“0000 0000”时，称为通用呼叫地址，用于寻访连接到 I^2C 总线上所有器件的地址。但是，如果器件在通用呼叫结构中不需要任何数据，它可以通过不发出响应来忽略这个地址。如果器件要求从广播呼叫地址得到数据，它会响应这个地址并作为从机接收器运转。第 2 个和接下来的字节会被能处理这些数据的每个从机接收器响应。通用呼叫地址的含意通常在第 2 个字节说明。格式如图 11-38 所示。

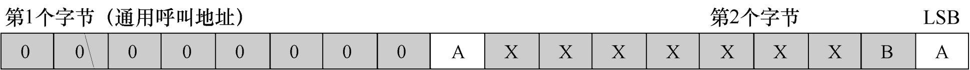


图 11-38　通用呼叫地址使用格式

当第 2 个字节的方向位 B 为“0”时，分以下两种情况：

第 2 个字节为 06H（00000110）时，硬件复位和通过硬件写入从机地址的可编程部分，即接收到这个两字节序列时，所有打算响应这个通用呼叫地址的器件将复位，并接受它们地址的可编程部分。要保证能响应命令的从机器件复位时不拉低 SDA 和 SCL 线，以免堵塞总线。

第 2 个字节为 04H（00000100）时，通过硬件写从机地址的可编程部分。所有通过硬件定义地址可编程部分的器件会在接收这个两字节序列时锁存可编程的部分，但器件不会复位。

当第 2 个字节的方向位 B 为“1”时，这个两字节序列是一个硬件通用呼叫，即序列由

一个硬件主机器件发送，例如，键盘扫描器不能编程来发送一个期望的从机地址。由于硬件主机预先不知道报文要传输给哪个器件，它只能产生这个硬件通用呼叫和它自己的地址让系统识别它，如图 11-39 所示。

S	0000　0000	A	主机地址	1	A	数据	A	数据	A	P

图 11-39　硬件通用呼叫格式

第 2 个字节中剩下的 7 位是硬件主机的地址。这个地址被一个连接到总线的智能器件识别（例如微控制器），并指引硬件主机的信息。如果硬件主机作为从机，它的从机地址和主机地址一样。

在一些系统中，可以选择系统复位时硬件主机器件工作在从机接收器方式，这时由系统中的主机告诉硬件主机器件数据应送往的从机器件地址，当硬件主机器件要发送数据时，就可以直接向指定机器发送数据了。

（3）起始字节

微控制器可以用两种方法连接到 I^2C 总线。有片上硬件 I^2C 总线接口的微控制器可被编程为只由总线的请求中断。当器件没有这种接口时，它必须经常通过软件监控总线。显然，微控制器监控或查询总线的次数越多，用于执行自己功能的时间越少。

因此，快速硬件器件和相关的依靠查询的慢速微控制器有速度差别。

此时，数据传输前应有一个比正常时间长的起始过程，如图 11-40 所示。起始过程包括起始条件（S）、起始字节（0000 0001）、响应时钟脉冲（ACK）和重复起始条件（Sr）组成。

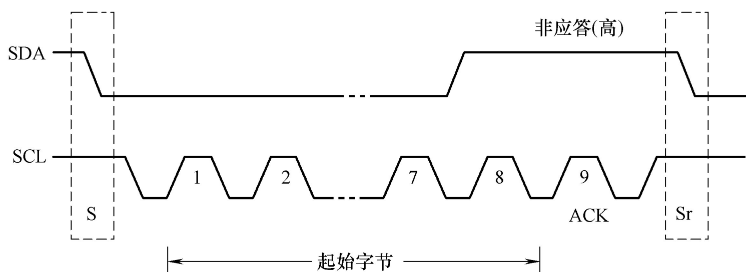


图 11-40　起始过程

在起始信号后的应答脉冲仅仅是为了和总线所使用的格式一样，并不要求器件在这个脉冲期间作应答。

11. 4. 3　AT24C 系列存储器的软硬件设计实例

1. AT24C 系列存储器介绍

AT24C 系列存储器是 ATMEL 公司的典型串行 E^2PROM 产品。24C × ×系列的引脚设置如图 11-41 所示。

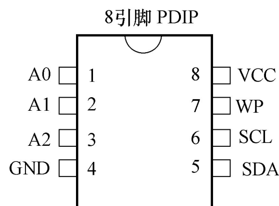


图 11-41　AT24C 系列 E^2PROM 引脚图

引脚介绍见表 11-7。

其型号与容量的关系为：

AT24C01：128B（128 ×8 位）；

AT24C02：256B（256 ×8 位）；

表 11-7　引脚介绍

引　脚	功　能	引　脚	功　能
A0 ~ A2	地址编码输入	WP	写保护
GND	接地	SCL	时钟信号输入
VCC	电源	SDA	数据信号

AT24C04：512B（512 ×8 位）；

AT24C08：1KB（1k ×8 位）；

AT24C16：2KB（2k ×8 位）。

（1）写入过程

AT24C 系列 E^2PROM 芯片的固定地址部分为 1010；可变的 3 位地址编码由 A2、A1、A0 引脚接高、低电平确定，形成一个 7 位的器件编码地址。

单片机进行写入操作时，首先发送该器件的 7 位地址码和写方向位“0”，发送完后释放 SDA 线并在 SCL 线上产生第 9 个时钟信号。被选中的存储器在确认是自己的地址后，在 SDA 线上产生一个应答信号作为响应，单片机收到应答后就可以传送数据了。写入 n 个字节的格式如图 11-42 所示。

传送数据时，单片机首先发送一个字节的被写入器件的存储区首地址。收到存储器的应答后，单片机就逐个发送各数据字节，但每发送一个字节都要等待应答。

AT24C 系列器件片内地址在接收到每一个数据字节后自动加 1，在芯片的字节限度内，只需输入首地址。在超过芯片的字节限度时，数据地址将返回到首地址，前面的数据将被覆盖。

当要写入的数据传送完后，单片机应发出终止信号以结束写入操作。



图 11-42　写入 n 个字节的格式

（2）读出过程

单片机进行读操作，首先发送该器件的 7 位地址码和写方向位“0”（“伪写”：装入要读出器件存储区的首地址），发送完后释放 SDA 线并在 SCL 线上产生第 9 个时钟信号。被选中的存储器件在确认是自己的地址后，在 SDA 线上产生一个应答信号作为响应。

然后，单片机再发送一个字节的要读出器件的存储区首地址，收到存储器的应答后，单片机再重复一次起始信号并发出器件地址和读方向位（“1”），收到器件应答后就可以读出数据字节。每读出一个字节，单片机都要回复应答信号。当最后一个字节数据读完后，单片机应返回一个“非应答”（高电平）信号，并发出终止信号以结束读出操作。

读出 n 个字节的数据格式如图 11-43 所示。



图 11-43　读出 n 个字节的数据格式

无论是写入还是读出，被操作在 256B 以内时，一个字节（8 位）的寻址范围即可满足要求。当容量大于 256B 时，采用占用器件引脚地址（A0、A1、A2）的办法，将引脚地址

作为页地址（占用的引脚地址线悬空）。当容量为 512B 时（2 页），A1、A2 作为器件地址编码，A0 引脚作为页码地址（悬空）；当容量为 1KB 时（4 页），A2 作为器件地址编码，A0、A1 引脚作为页码地址（悬空）；当容量为 2KB 时（8 页），A0、A1、A2 引脚都作为页码地址（悬空）；当容量在 4KB 以上时，存储单元地址字节将用两个字节表示。

2. 读/写 AT24C 系列存储器应用硬件原理图

本例中将介绍时序模拟 I^2C 总线读/写 AT24C 系列存储器的应用设计参考。

硬件原理图如图 11-44 所示。

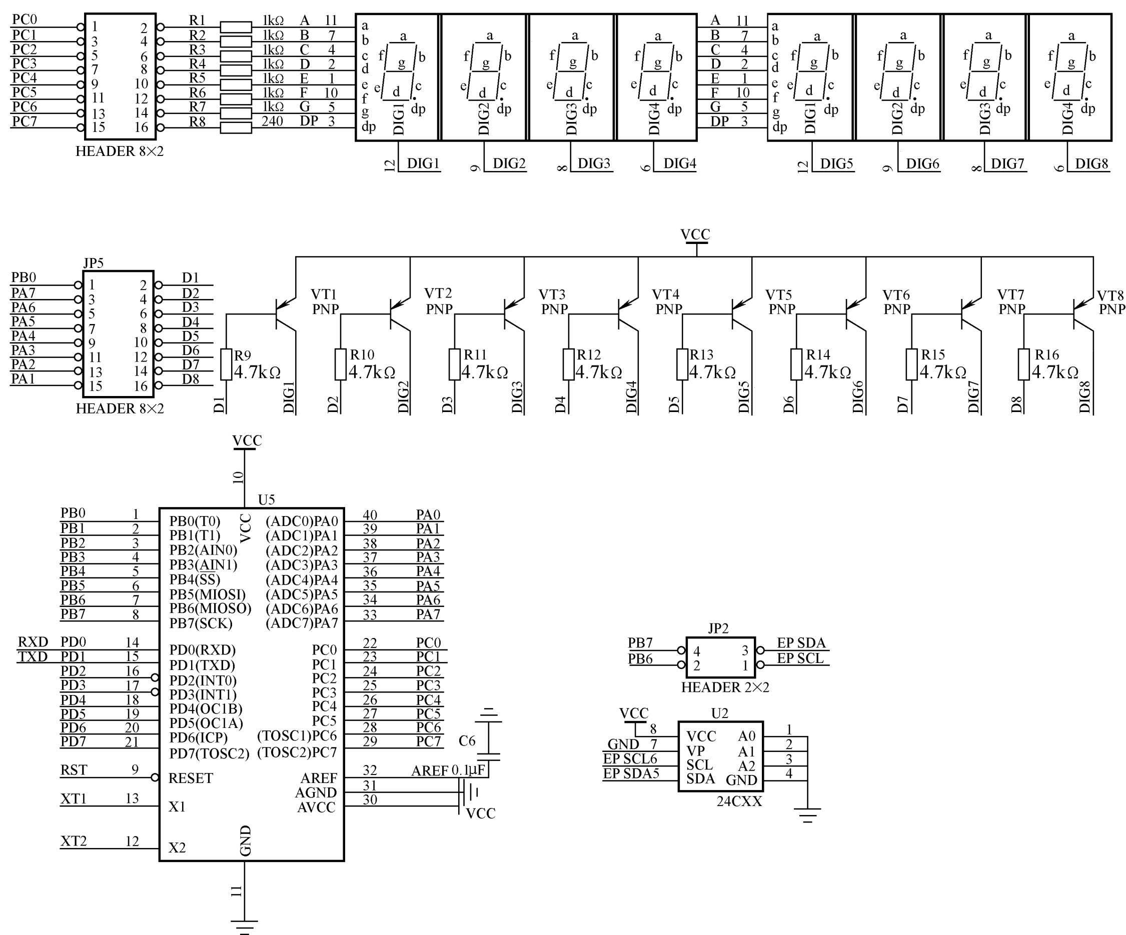


图 11-44　读/写 AT24C 系列存储器原理图

3. 软件设计

```
/******************************************************/
/* EEPROM 端口模拟 I2C 测试程序                          */
/* 目标器件：ATmega16                                    */
/* 晶振：RC 1MHz                                         */
/* 编译环境：ICCAVR 6.31A                                */
/******************************************************/
#include <iom16v.h>
```

```
#include <macros. h>

/************************数据定义************************/
#define OP_READ0xa1                // 器件地址以及读取操作
#define OP_WRITE 0xa0              // 器件地址以及写入操作

/************************端口定义************************/

#define        SCLOn        PORTB |=(1 <<6)            //时钟高
#define        SCLOff       PORTB &= ~(1 <<6)
#define        SDAOn        PORTB |=(1 <<7)            //数据高
#define        SDAOff       PORTB &= ~(1 <<7)
#define        SDAOut       DDRB|=(1 <<7)
#define        SDAIn        DDRB&= ~(1 <<7)

/***********************数码管段码表***********************/
extern const unsigned char tab[ ] = { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,
0x90,
              /*   0    1    2    3    4    5    6    7    8    9   */
                            0x88,0x83,0xC6,0xA1,0x86,0x8E};
              /*   a    b    c  d    e    f   */

/***********************定义全局变量***********************/
int eepromdata;                  //从 EEPROM 里读出来的数据

/*******************************************************
函数功能:延时子程序
入口参数:
出口参数:
*******************************************************/
void delay(void)
{
    int k;
    for(k =0;k <400;k + +);
}

/*******************************************************
函数功能:数码管延时程序
入口参数:
出口参数:
```

```
***********************************************************/
void delay_1us(void)                    //1μs 延时函数
{
    asm("nop");
}

void delay_nus(unsigned int n)          //Nμs 延时函数
{
    unsigned int i=0;
    for(i=0;i<n;i++)
    delay_1us();
}

void delay_1ms(void)                    //1ms 延时函数
{
    unsigned int i;
    for(i=0;i<140;i++);
}

void delay_nms(unsigned int n)          //N ms 延时函数
{
    unsigned int i=0;
    for(i=0;i<n;i++)
    delay_1ms();
}


/***********************************************************
函数功能:开始信号
入口参数:
出口参数:
***********************************************************/
void start()
{
    SDAOn;
    delay_nus(5);
    SCLOn;
    delay_nus(5);
    SDAOff;
```

```
    delay_nus(5);
    SCLOff;
}

/*******************************************************
函数功能:停止信号
入口参数:
出口参数:
*******************************************************/
void stop()
{
    SDAOff;
    delay_nus(5);
    SCLOn;
    delay_nus(5);
    SDAOn;

}

/*******************************************************
函数功能:读取数据
入口参数:
出口参数:read_data
*******************************************************/
unsigned char shin()
{
    unsigned char i,read_data;
    SDAIn;          //输入
    for(i=0; i< 8; i++)
    {
        delay_nus(5);
        SCLOff;
        delay_nus(5);
        SCLOn;
        delay_nus(5);
        read_data <<=1;
        if(PINB&0x80)
          read_data|=0x01;
        else
          read_data&=0xfe;
```

```
    }
    SDAOut;        //输出
    return(read_data);
}

/*************************************************************
函数功能:向 EEPROM 写数据
入口参数:write_data
出口参数:ack_bit
*************************************************************/
unsigned char shout(unsigned char write_data)
{
    unsigned char i,ack_bit;
    for(i=0; i< 8; i++)
    {
        if(write_data & 0x80)
        {
            SDAOn;
        }
        else
        {
            SDAOff;
        }
        SCLOn;
        delay_nus(5);
        SCLOff;
        write_data <<=1;
    }
    SDAOn;
    delay_nus(5);
    SCLOn;
    delay_nus(5);
    if(PINB&0x80)
      ack_bit=1;
    else
      ack_bit=0;
    SCLOff;
    delay_nus(5);
    return ack_bit;          //返回 AT24Cxx 应答位
}
```

```
/ ************************************************************
函数功能:向指定地址写数据
入口参数:addr,write_data
出口参数:
************************************************************ /
void write_byte(unsigned char addr,unsigned char write_data)
{
      start();
      shout(OP_WRITE);
      shout(addr);
      shout(write_data);
      stop();
      delay_nms(10);
}

/ ************************************************************
函数功能:读取当前地址数据
入口参数:
出口参数:read_data
************************************************************ /
unsigned char read_current()
{
      unsigned char read_data;
      start();
      shout(OP_READ);
      read_data = shin();
      stop();
      return read_data;
}

/ ************************************************************
函数功能:向指定地址读数据
入口参数:random_addr
出口参数:read_data
************************************************************ /
unsigned char read_random(unsigned char random_addr)
{
      start();
      shout(OP_WRITE);
```

```
    shout(random_addr);
    return(read_current());
}

/*********************************************************
函数功能:显示子程序
入口参数:k
出口参数:
*********************************************************/
void display(unsigned char k)//显示十六进制
{
    PORTC = tab[k/16];
    PORTA = 0xf7;
    delay_nms(2);

    PORTC = tab[k%16];
    PORTA = 0xef;
    delay_nms(2);
}


/*********************************************************
函数功能:主程序
入口参数:
出口参数:
*********************************************************/
void main(void)
{
    DDRC = 0xff;
    PORTC = 0xff; //LED Data
    DDRA = 0xff;
    PORTA = 0xff; //LED Contrl
    DDRB = 0xff;   //SDA = 1;
    PORTB = 0xff; //SCL = 1;

    display(0);
    eepromdata = 0;
    write_byte(0x01,0x55);          //向 0x01 地址写入 0x55(85)的数据
    delay_nms(50);
    write_byte(0x02,0xAA);          //向 0x02 地址写入 0xAA(170)的数据
```

```
        delay_nms(50);
        delay_nms(50);
        eepromdata = read_random(0x02);            //读取其中一个地址内的数据来验证
        delay_nms(50);
        while(1)
        {
            display(eepromdata);
        }
}
```

11.5　93CXX 系列存储器应用实例

前一节介绍了 I²C 总线结构的 24CXX 系列存储器的应用，在一般的存储器件中 SPI 结构的存储器也用得比较广泛，本节介绍 SPI 总线结构的 93CXX 系列存储器的应用。

11.5.1　SPI 总线简介

1. SPI 总线基本概念

SPI（Serial Peripheral Interface，串行外设接口）总线是 Motorola 公司推出的一种同步串行接口技术。SPI 总线系统是一种同步串行外设接口，允许 MCU 与各种外围设备以串行方式进行通信、数据交换。外围设备包括 FLASHRAM、A/D 转换器、网络控制器、MCU 等。SPI 是一种高速的、全双工、同步的通信总线，并且在芯片的引脚上只占用 4 根线，节约了芯片的引脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议。其工作模式有两种：主模式和从模式。SPI 是一种允许一个主设备启动一个从设备同步通信的协议，从而完成数据的交换。也就是 SPI 是一种规定好的通信方式。这种通信方式的优点是占用端口较少，一般 4 根就够基本通信了（不算电源线），同时传输速度也很高。一般来说要求主设备要有 SPI 控制器（也可用模拟方式），就可以与基于 SPI 的芯片通信了。

2. SPI 总线系统结构

SPI 系统可直接与各个厂商生产的多种标准外围器件直接接口，一般使用 4 条线：串行时钟线（SCK）、主机输入/从机输出数据线 MISO（DO）、主机输出/从机输入数据线 MOSI（DI）和低电平有效的从机选择线（CS）。MISO 和 MOSI 用于串行接收和发送数据，先为 MSB（高位），后为 LSB（低位）。在 SPI 设置为主机方式时，MISO 是主机数据输入线，MOSI 是主机数据输出线。SCK 用于提供时钟脉冲将数据一位位地传送。SPI 总线器件间传送数据框图如图 11-45 所示。

3. SPI 总线的接口特性

利用 SPI 总线可在软件的控制下构成各种系统。如 1 个主 MCU 和几个从 MCU、几个从 MCU 相互连接构成多主机系统（分布式系统）、1 个主 MCU 和 1 个或几个从 I/O 设备所构成的各种系统等。在大多数应用场合，可使用 1 个 MCU 作为主控机来控制数据，并向 1 个或几个从外围器件传送该数据。从器件只有在主机发命令时才能接收或发送数据。其数据的传输格式是高位（MSB）在前，低位（LSB）在后。

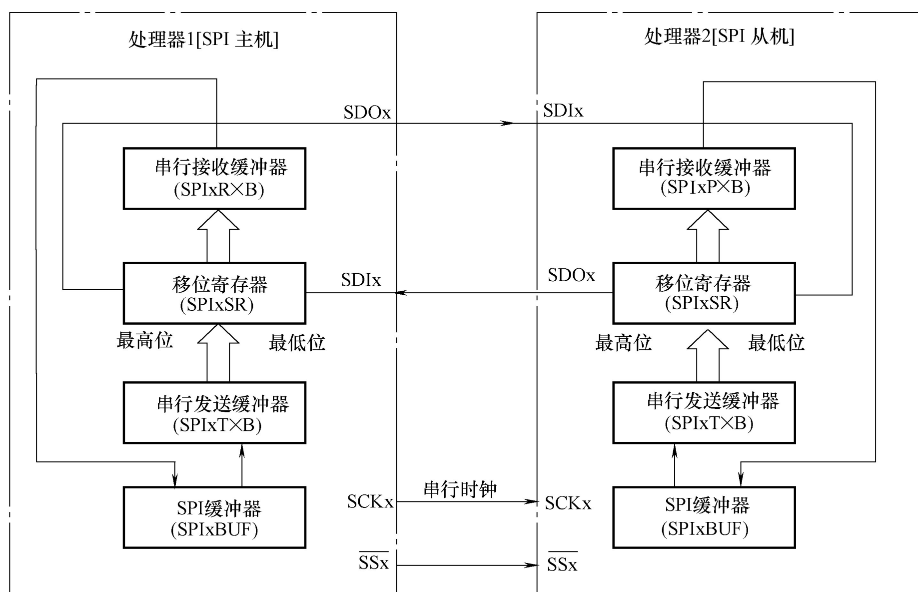


图 11-45　SPI 总线器件间传送数据框图

当一个主控机通过 SPI 与几种不同的串行 I/O 芯片相连时，必须使用每片的允许控制端，这可通过 MCU 的 I/O 端口输出线来实现。但应特别注意这些串行 I/O 芯片的输入输出特性：首先是输入芯片的串行数据输出是否有三态控制端。平时未选中芯片时，输出端应处于高阻态。若没有三态控制端，则应外加三态门。否则 MCU 的 MISO 端只能连接 1 个输入芯片。其次是输出芯片的串行数据输入是否有允许控制端。因为只有在此芯片允许时，SCK 脉冲才把串行数据移入该芯片；在禁止时，SCK 对芯片无影响。若没有允许控制端，则应在外围用门电路对 SCK 进行控制，然后再加到芯片的时钟输入端；当然，也可以只在 SPI 总线上连接 1 个芯片，而不再连接其他输入或输出芯片。

4. SPI 总线的数据传输

SPI 是一个环形总线结构，其时序其实很简单，主要是在 SCK 的控制下，两个双向移位寄存器进行数据交换。SPI 数据传输原理很简单，它需要至少 4 根线，事实上 3 根也可以。也是所有基于 SPI 的设备共有的，它们是 SDI（数据输入）、SDO（数据输出）、SCK（时钟）、CS（片选）。其中 CS 是控制芯片是否被选中，也就是说只有片选信号为预先规定的使能信号时（高电位或低电位），对此芯片的操作才有效。这就允许在同一总线上连接多个 SPI 设备成为可能。在 SPI 方式下数据是一位一位地传输的。这就是 SCK 时钟线存在的原因，由 SCK 提供时钟脉冲，SDI、SDO 则基于此脉冲完成数据传输。数据输出通过 SDO 线，数据在时钟上沿或下沿时改变，在紧接着的下沿或上沿被读取，完成一位数据传输，输入也使用同样原理。这样，在至少 8 次时钟信号的改变（上沿和下沿为一次），就可以完成 8 位数据的传输。假设 8 位寄存器内装的是待发送的数据 10101010，上升沿发送、下降沿接收、高位先发送。那么第 1 个上升沿来的时候 数据将会是高位数据 SDO = 1。下降沿到来的时候，SDI 上的电平将被存到寄存器中去，那么这时寄存器 = 0101010SDI，这样在 8 个时钟脉冲以后，两个寄存器的内容互相交换一次。这样就完成里一个 SPI 时序。下面举一个实例来说明其数据传送过程。

假设主机和从机初始化就绪，并且主机的 sbuff = 0xaa，从机的 sbuff = 0x55，表 11-8 将分步对 SPI 的 8 个时钟周期的数据情况演示一遍（表中“上”表示上升沿，“下”表示下降沿）。

表 11-8　脉冲与数据变化对应表

脉冲序号	主机缓存	从机缓存	SDI	SDO
0	10101010	01010101	0	0
1 上	0101010x	1010101x	0	1
1 下	01010100	10101011	0	1
2 上	1010100x	0101011x	1	0
2 下	10101001	01010110	1	0
3 上	0101001x	1010110x	0	1
3 下	01010010	10101101	0	1
4 上	1010010x	0101101x	1	0
4 下	10100101	01011010	1	0
5 上	0100101x	1011010x	0	1
5 下	01001010	10110101	0	1
6 上	1001010x	0110101x	1	0
6 下	10010101	01101010	1	0
7 上	0010101x	1101010x	0	1
7 下	00101010	11010101	0	1
8 上	0101010x	1010101x	1	0
8 下	01010101	10101010	1	0

这样就完成了两个寄存器 8 位的交换，SDI、SDO 是相对于主机而言的。其中 CS 引脚作为主机的时候，从机可以把它拉低被动选为从机，作为从机的时候，可以作为片选脚用。根据以上分析，一个完整的传送周期是 16 位，即两个字节，因为首先主机要发送命令过去，然后从机根据主机的命令准备数据，主机在下一个 8 位时钟周期才把数据读回来。这样的传输方式有一个优点，与普通的串行通信不同，普通的串行通信一次连续传送至少 8 位数据，而 SPI 允许数据一位一位地传送，甚至允许暂停，因为 SCK 时钟线由主控设备控制，当没有时钟跳变时，从设备不采集或传送数据。也就是说，主设备通过对 SCK 时钟线的控制可以完成对通信的控制。SPI 还是一个数据交换协议，因为 SPI 的数据输入和输出线独立，所以允许同时完成数据的输入和输出。

对于不带 SPI 串行总线接口的单片机来说，可以使用软件来模拟 SPI 的操作，包括串行时钟、数据输入和数据输出。如可以定义 3 个普通 I/O 口用来模拟 SPI 器件的 SCK、MISO、MOSI。对于不同的串行接口外围芯片，它们的时钟时序是不同的。对于在 SCK 的上升沿输入（接收）数据和在下降沿输出（发送）数据的器件，一般应将其串行时钟输出口的初始状态设置为 1，而在允许接口后再置为 0。这样，MCU 在输出 1 位 SCK 时钟的同时，将使接口芯片串行左移，从而输出 1 位数据至单片机的模拟 MISO 线，此后再置 SCK 为 1，使单片机从模拟的 MOSI 线输出 1 位数据（先为高位）至串行接口芯片。至此，模拟 1 位数据输入输出便宣告完成。此后再置 SCK 为 0，模拟下 1 位数据的输入输出……依此循环 8 次，即可完成 1 次通过 SPI 总线传输 8 位数据的操作。对于在 SCK 的下降沿输入数据和上升沿输出数据的器件，则应取串行时钟输出的初始状态为 0，即在接口芯片允许时，先置 SCK 为 1，以便外围接口芯片输出 1 位数据（MCU 接收 1 位数据），之后再置时钟为 0，使外围接口芯片

接收 1 位数据（MCU 发送 1 位数据），从而完成 1 位数据的传送。

11.5.2　93C46 存储器的软硬件设计实例

下面就以目前单片机系统中广泛应用的 SPI 接口的 93C46 存储器为例，介绍 SPI 器件的基本应用。

1. 93C46 串行存储器简介

93C46 是 1kbit 串行 EEPROM 储存器。每一个储存器都可以通过 DI/DO 引脚写入或读出。它的存储容量为 1024 位，内部为 128 × 8 位或 64 × 16 位。93C46 为串行三线 SPI 操作芯片，在时钟时序的同步下接收数据口的指令。指令码为 9 位十进制码，具有 7 个指令，读、擦写使能、擦除、写、全擦、全写及擦除禁止。该芯片擦写时间快，有擦写使能保护，可靠性高，擦写次数可达 100 万次，93C46 的引脚功能图如图 11-46 所示。93C46 串行 E^2PROM 指令格式选择见表 11-9。

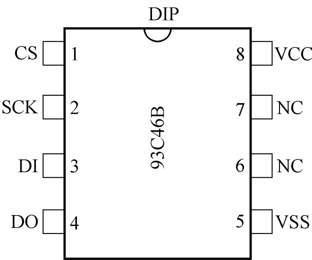


图 11-46　93C46 的引脚图

表 11-9　93C46 串行 EEPROM 指令格式选择表

指　令	起始位	操作数	地　址		数　据	
			64 × 16	128 × 8	64 × 16	128 × 8
读(READ)	1	10	A5 ~ A0	A6 ~ A0		
清除(ERASE)	1	11	A5 ~ A0	A6 ~ A0		
写(WRITE)	1	01	A5 ~ A0	A6 ~ A0	D15 ~ D0	D7 ~ D0
写使能(EWEN)	1	00	11XXXX	11XXXXX		
写禁止(EWDS)	1	00	00XXXX	00XXXXX		
芯片清除(ERAL)	1	00	10XXXX	10XXXXX		
芯片写入(WRAL)	1	00	01XXXX	01XXXXX	D15 ~ D0	D7 ~ D0

指令说明：

1）读（READ）：当下达 10XXXXXX 指令后，地址（XXXXXXXX）的数据在 SCK = 1 时由 DO 输出。

2）写（WRITE）：在写入数据前，必须先下达写使能（EWEN）指令，然后再下达 01XXXXXX 指令后，当 SCK = 1 时，会把数据码写入指定地址（XXXXXXXX）；而 DO = 0 时，表示还在进行写操作，写入结束后 DO 会转为高电平。写入动作完成后，必须再下达写禁止（EWDS）命令。

3）清除（ERASE）：下达清除指令 11XXXXXX 后会将地址（XXXXXXXX）的数据清除。

4）写使能（EWEN）：下达 0011XXXX 指令后，才可以进行写（WRITE）操作。

5）写禁止（EWDS）：下达 0000XXXX 指令后，才可重复进行写入（WRITE）操作。

6）芯片清除（ERAL）：下达 0010XXXX 指令后，全部禁止。

7）芯片写入（WRAL）：下达 0001XXXX 指令后，全部写入“0”。

2. 程序功能

本例用来实现对 93C46 存储器的读写操作，并验证数据是否正确。本程序先分别向 0x02 和 0x03 两个地址写入 0x55 和 0xAA，然后读其中一个地址，并将读到的数据显示出来验证是否正确。程序默认是读 0x02 地址内的数据，读者也可以修改地址数据来读其他地址数据。

3. 硬件原理图（见图 11-47）

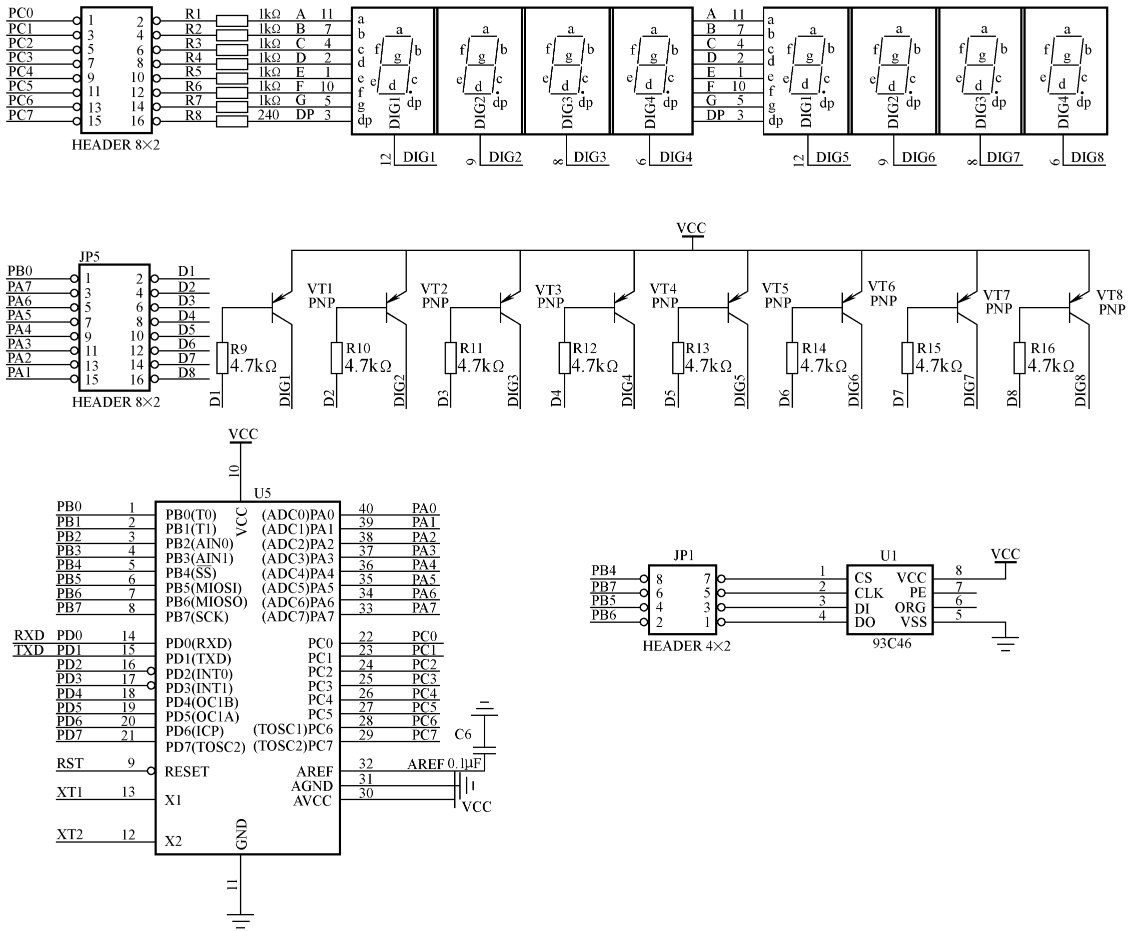


图 11-47　硬件原理图

4. 软件设计

```
/ ************************************************************************ /
/ * SPI 93C46 端口模拟测试程序                                             * /
/ * 目标器件:ATmega16                                                     * /
/ * 晶振:RC 1MHz                                                          * /
/ * 编译环境:ICCAVR 6.31A                                                 * /
/ ************************************************************************ /

/ ***************************** 包含头文件 ***************************** /
#include  <iom16v.h>
#include  <macros.h>

/ ***************************** 端口定义 ***************************** /
#define   CS      4
```

```
#define  SK     7
#define  DI     5
#define  DO     6

#define  DI_H    PORTB |=(1<<DI)
#define  DI_L    PORTB &=~(1<<DI)
#define  DO_H    PORTB |=(1<<DO)
#define  DO_L    PORTB &=~(1<<DO)
#define  CS_H    PORTB |=(1<<CS)
#define  CS_L    PORTB &=~(1<<CS)
#define  SK_H    PORTB |=(1<<SK)
#define  SK_L    PORTB &=~(1<<SK)

/***************************** 数码管段码表 *****************************/
extern const unsigned char tab[ ] = { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,
0x80,0x90,
            /*   0    1    2    3    4    5    6    7    8    9  */
                 0x88,0x83,0xC6,0xA1,0x86,0x8E};
            /*   a    b    c  d    e    f  */

/***************************** 定义全局变量 *****************************/
int readdata;            //从 93C46 读出的数据
unsigned char temp=0,InData=0;
unsigned int  result = 0;

/*************************************************************************
函数功能:数码管延时程序
入口参数:
出口参数:
*************************************************************************/
void delay_1us(void)                //1μs 延时函数
{
asm("nop");
}

void delay_nus(unsigned int n)      //N μs 延时函数
{
unsigned int i=0;
for(i=0;i<n;i++)
delay_1us();
```

```
}

void delay_1ms(void)                    //1ms 延时函数
{
unsigned int i;
for(i=0;i<140;i++);
}

void delay_nms(unsigned int n)        //N ms 延时函数
{
unsigned int i=0;
for(i=0;i<n;i++)
delay_1ms();
}

/**********************************************************************
函数功能:显示子程序
入口参数:k
出口参数:
**********************************************************************/
void display(unsigned int k)
{
      PORTC=tab[k/1000];
      PORTA=0xef;
      delay_nms(2);

      PORTC=tab[k/100%10];
      PORTA=0xf7;
      delay_nms(2);

      PORTC=tab[k/10%10];
      PORTA=0xfb;
      delay_nms(2);

      PORTC=tab[k%10];
      PORTA=0xfd;
      delay_nms(2);
}

/**********************************************************************
```

```
函数功能:写入数据使能子程序
入口参数:
出口参数:
************************************************************************/
void Ewen(void)
{
  CS_L;
  SK_L;
  CS_H;
  InData = 0x98;                                    //写使能指令 10011XXXX
  for(temp = 9;temp!  = 0;temp--)
  {
    if(InData&0x80)
      DI_H;
    else
      DI_L;
    SK_H;
    SK_L;
    InData < < = 1;
  }
  CS_L;
}

/************************************************************************
函数功能:写入数据禁止子程序
入口参数:
出口参数:
************************************************************************/
void Ewds(void)//擦除/写禁止
{
  CS_L;
  SK_L;
  CS_H;
  InData = 0x80;                                    //写禁止指令 10000XXXX
  for(temp = 9;temp! = 0;temp--)
  {
    if(InData&0x80)
      DI_H;
    else
      DI_L;
```

```
      SK_H;
      SK_L;
      InData < < =1;
    }
    CS_L;
}

/ ***********************************************************************
函数功能:数据清除子程序
入口参数:
出口参数:
*********************************************************************** /
void Erase(unsigned char address)
{
    Ewen();
    SK_L;
    DI_H;                                              //起始位 1
    CS_L;
    CS_H;
    SK_H;
    SK_L;                                              //起始位结束
    address| =0xc0;
    for(temp =8;temp!  =0;temp--)                        //写入 8 位清除指令
    {
      if(address&0x80)
        DI_H;
      else
        DI_L;
      SK_H;
      SK_L;
      address < < =1;
    }
    CS_L;
    DO_H;
    CS_H;
    SK_H;
    while((PINB&0x40) = =0)                              //检测写入是否结束
    {
      SK_L;
      SK_H;
```

```
    }
    SK_L;
    CS_L;
    Ewds( );
}

/ ************************************************************************
函数功能:芯片写入子程序
入口参数:
出口参数:
************************************************************************ /
void Wral( unsigned int _InData)
{
    unsigned char address = 0x88;                        //芯片写入指令 10001XXXX
    Ewen( );
    CS_L;
    SK_L;
    CS_H;

    for( temp = 9; temp!  = 0; temp-- )                  //写入芯片写入指令
    {
      if( address&0x80)
        DI_H;
      else
        DI_L;
      SK_H;
      SK_L;
      address < < = 1;
    }
    for( temp = 16; temp!  = 0; temp-- )                 //写入 16 位数据码,全为 0
    {
      if( _InData&0x80)
        DI_H;
      else
        DI_L;
      SK_H;
      SK_L;
      _InData < < = 1;
    }
    CS_L;
```

```
    DO_H;
    CS_H;
    SK_H;
    while((PINB&0x40) = =0)                              //检测写入是否结束
    {
      SK_L;
      SK_H;
    }
    SK_L;
    CS_L;
    Ewds();
}

/ *************************************************************************
函数功能:芯片清除子程序
入口参数:
出口参数:
 ************************************************************************* /
void Eral(void)
{
    Ewen();
    CS_L;
    SK_L;
    CS_H;
    InData =0x90;                                        //清除楚指令 10010XXXX
    for(temp =9;temp!  =0;temp--)                         //写入清除指令
    {
      if(InData&0x80)
        DI_H;
      else
        DI_L;
      SK_H;
      SK_L;
      InData < < =1;
    }
    CS_L;
    DO_H;
    CS_H;
    SK_H;
    while((PINB&0x40) = =0)                              //检测写入是否结束
```

```
    {
        SK_L;
        SK_H;
    }
    SK_L;
    CS_L;
    Ewds();
}

/ *****************************************************************
函数功能:读出某地址数据子程序
入口参数:address
出口参数:result
 ***************************************************************** /
unsigned int Read(unsigned char address)//读
{
    Ewen();
    SK_L;
    DI_H;                                       //起始位 1
    CS_L;
    CS_H;
    SK_H;
    SK_L;                                       //起始位结束
    address = address&0x3f|0x80;                //bit7 置 1,bit6 清零
    for(temp = 8;temp! =0;temp--)
    {
        if(address&0x80)
            DI_H;
        else
            DI_L;                               //写入 8 位地址码
        SK_H;
        SK_L;
        address < < =1;
    }
    DO_H;                                       //写入结束
    for(temp = 16;temp! =0;temp--)              //读出 16 位数据码
    {
        SK_H;
        result < < =1;
        if(PINB&0x40)
```

```
            result| =0x01;
        else
            result& =0xfe;
        SK_L;
    }
    CS_L;
    Ewds();
    return(result);
}

/ *********************************************************************
函数功能:写入数据子程序
入口参数:address,_InData
出口参数:
********************************************************************* /
void Write(unsigned char address,unsigned int _InData)
{
    Ewen();
    SK_L;
    DI_H;                                          //起始位 1
    CS_L;
    CS_H;
    SK_H;
    SK_L;                                          //起始位结束
    address = address&0x3f|0x40;                   //bit7 清零;bit6 置 1
    for(temp =8;temp!  =0;temp--)                  //写入 8 位地址码
    {
        if(address&0x80)
            DI_H;
        else
            DI_L;
        SK_H;
        SK_L;
        address < < =1;
    }
    for(temp =16;temp!  =0;temp--)                 //写入 16 位数据码
    {
        if(_InData&0x8000)
            DI_H;
        else
```

```
        DI_L;
      SK_H;
      SK_L;
      _InData < < =1;
    }
    CS_L;
    DO_H;
    CS_H;
    SK_H;
    while((PINB&0x40) = =0)          //检测写入是否结束,写入结束后 DO 为高电平
    {
      SK_L;
      SK_H;
    }
    SK_L;
    CS_L;
    Ewds();
}

/ ************************************************************************
函数功能:主程序
入口参数:
出口参数:
************************************************************************ /
void main(void)
{
    unsigned char i;
    DDRC =0xff;
    PORTC =0xff;
    DDRA =0xff;
    PORTA =0xff;

    DDRB  = 0xbe;
    CS_L;
    SK_L;
    DI_H;
    DO_H;

    display(0);
    Ewen();                //使能写入操作
```

```
    Eral( );                    //擦除全部内容
    Write(0x02, 0x55);          //向 0x02 地址写入 0x55(85)

    while(1)
    {
      readdata = Read(0x02);    //读取其中一个地址内数据验证
      display(readdata);        //显示数据
    }
}
```

11.6 DS1302 时钟芯片应用实例

在很多单片机系统中都要求带有实时时钟电路，如最常见的数字钟、钟控设备、数据记录仪表，这些仪表往往需要采集带时标的数据，同时一般它们也会有一些需要保存起来的重要数据，有了这些数据，便于用户后期对数据进行观察、分析。本节就介绍市面上常见的时钟芯片 DS1302 的应用。DS1302 是美国 DALLAS 公司推出的一款高性能、低功耗、带内部 RAM 的实时时钟芯片（RTC），也就是一种能够为单片机系统提供日期和时间的芯片。本节将会把 RTC 相关的一些技术粗略介绍一下，然后介绍 DS1302 与单片机之间的软硬件应用。

11.6.1 实时时钟（RTC）简介

实时时钟芯片的主要功能是完成年、月、周、日、时、分、秒的计时，通过外部接口为单片机系统提供日历和时钟，所以一个最基本的实时时钟芯片通常会具有如下的一些部件：电源电路、时钟信号产生电路、实时时钟、数据存储器、通信接口电路、控制逻辑电路等，如图 11-48 所示，同时大部分的 RTC 还会提供一些额外的 RAM。

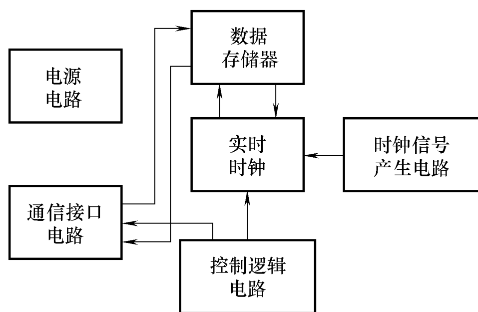


图 11-48 RTC 的基本组成

如果直接利用单片机的定时器，是不是也可以用软件自己来写时钟、日历程序？是的，但是会有几个问题，首先为了使时钟不至于停走，就得在停电时给单片机供电，而相对 RTC 来说，单片机的功耗大很多，电池往往无法长时间工作；其次单片机计时的准确度比较差，通常很难达到需要的精度，因此目前 RTC 的使用已经十分广泛。

由于在需要 RTC 的场合一般不允许时钟停走，所以即使在单片机系统停电时，RTC 也必须能正常工作，因此一般都需要电池供电，同时考虑到电池使用寿命，所以有不少 RTC 把电源电路设计成能够根据主电源电压自动切换的形式，自动切换 RTC 使用主电源或备用电池，即当断电的时候，后备电池能够自动给 RTC 供电，而像 DS1302 还增加了电池充电电路，用来对可充锂电池充电。

综上所述，RTC 电路的主要特点是功耗低、精度高。那么 RTC 在使用过程中是如何控制精度的呢？一般，RTC 都使用 32768Hz 的晶振，本身误差小（5PPM ~20PPM），同时很多设备在生产过程中对这个频率进行过校准，主要方法就是改变两个从晶振引脚到地的电容值的大小，通过测试 RTC 输出的秒信号的频率，然后把电容改成合适的数值，使精度控制在合理的范围里，当然目前也有些时钟芯片在片内内置了电容阵列，可以自动调整。影响精度还有另外一个原因，就是温度，因此有很多产品在采用无内置温补电路的时候，会使用软件对计时进行温度补偿。当然，现在也有些 RTC 内置了温度补偿，甚至还可以为系统提供环境温度值。

最常见的 RTC 可能是 DS1302 和 DS12887 了，当然其实还有很多其他的同类产品，表 11-10 按功能不同对几个也比较常见的 RTC 予以简单的比较。

表 11-10　一些常用 RTC 的功能比较

RTC 型号	生产商	接口方式	晶振内置	补偿方式	温度补偿	电池内置	充电电路	报警输出
DS12887	DALLAS	并行	是	无	无	是	有	有
DS1302	DALLAS	串行	否	无	无	否	有	无
DS3231	DALLAS	串行	是	硬件	有	否	无	有
RX8025	EPSON	串行	是	软件	无	否	无	有
PCF8563	PHILIPS	串行	否	无	无	否	无	有

1. DS1302 时钟芯片简介

DS1302 是 DALLAS 公司推出的涓流充电时钟芯片，内含一个实时时钟/日历和 31 字节静态 RAM，可以通过串行接口与单片机进行通信。实时时钟/日历电路提供秒、分、时、日、星期、月、年的信息，每个月的天数和闰年的天数可自动调整，时钟操作可通过 AM/PM 标志位决定采用 24 小时或 12 小时时间格式。DS1302 与单片机之间能简单地采用同步串行的方式进行通信，仅需 3 根 I/O 线：复位（RST）、I/O 数据线、串行时钟（SCLK）。时钟/RAM 的读/写数据以 1 字节或多达 31 字节的字符组方式通信。DS1302 工作时功耗很低，保持数据和时钟信息时，功耗小于 1mW。DS1302 主要性能如下：

1）实时时钟具有能计算 2100 年之前的秒、分、时、日、星期、月、年的能力及闰年调整的能力。

2）31 ×8 位暂存数据存储 RAM。

3）串行 I/O 口方式，引脚数量少。

4）宽电压工作范围：2.0 ~5.5V。

5）工作电流：2.0V 时小于 300nA。

6）读/写时钟或 RAM 数据时，有两种传送方式：单字节传送和多字节传送。

7）8 脚 DIP 封装或 SOIC 封装

2. DS1302 的内部结构

DS1302 的封装图如图 11-49 所示。

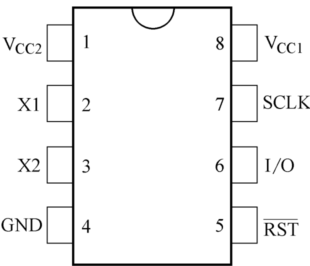


图 11-49　DS1302 封装图

DS1302 的内部结构如图 11-50 所示，主要组成部分为输入移位寄存器、命令与控制逻辑、振荡电路与分频器、实时时钟以及 RAM。虽然数据分成两种，但是对单片机的程序而

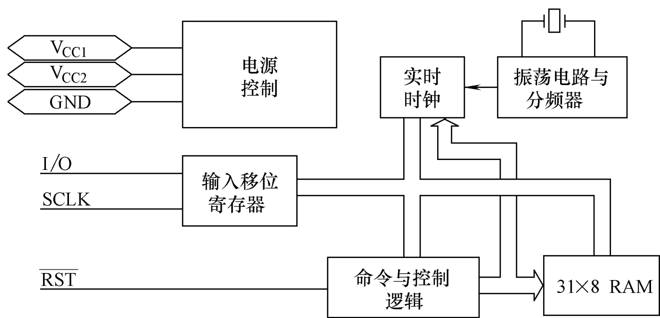


图 11-50　DS1302 的内部结构图

言，其实是一样的，就是对特定的地址进行读写操作。

DS1302 含充电电路，可以对作为后备电源的可充电电池充电，并可选择充电使能和串入的二极管数目，以调节电池充电电压。不过对目前而言，最需要熟悉的是与时钟相关部分的功能，对于其他参数请参阅数据手册。

3. DS1302 的工作原理

DS1302 工作时为了对任何数据传送进行初始化，需要将复位脚（RST）置为高电平，且将 8 位地址和命令信息装入移位寄存器。数据在时钟（SCLK）的上升沿串行输入，前 8 位指定访问地址，命令字装入移位寄存器后，在之后的时钟周期，读操作时输入数据，写操作时输入数据。时钟脉冲的个数在单字节方式下为 8 + 8（8 位地址 + 8 位数据），在多字节方式下为 8 加最多可达 248 位的数据。

4. DS1302 的寄存器和控制命令

对 DS1302 的操作就是对其内部寄存器的操作，DS1302 内部共有 12 个寄存器，其中有 7 个寄存器与日历、时钟相关，存放的数据位为 BCD 码形式。此外，DS1302 还有年份寄存器、控制寄存器、充电寄存器、时钟突发寄存器及与 RAM 相关的寄存器等。时钟突发寄存器可一次性顺序读写除充电寄存器以外的寄存器。日历、时间寄存器及控制字见表 11-11。

表 11-11　日历、时间寄存器与控制字对照表

寄存器名称	7	6	5	4	3	2	1	0
	1	RAM/CK	A4	A3	A2	A1	A0	RD/W
秒寄存器	1	0	0	0	0	0	0	
分寄存器	1	0	0	0	0	0	1	
小时寄存器	1	0	0	0	0	1	0	
日寄存器	1	0	0	0	0	1	1	
月寄存器	1	0	0	0	1	0	0	
星期寄存器	1	0	0	0	1	0	1	
年寄存器	1	0	0	0	1	1	0	
写保护寄存器	1	0	0	0	1	1	1	
慢充电寄存器	1	0	0	1	0	0	0	
时钟突发寄存器	1	0	1	1	1	1	1	

最后一位 RD/W 为“0”时表示进行写操作，为“1”时表示读操作。

DS1302 内部寄存器列表见表 11-12。

表 11-12　DS1302 内部主要寄存器分布表

寄存器名称	命令字		取值范围	各位内容							
	写	读		7	6	5	4	3	2	1	0
秒寄存器	80H	81H	00～59	CH	10SEC			SEC			
分寄存器	82H	83H	00～59	0	10MIN			MIN			
小时寄存器	84H	85H	01～12 或 00～23	12/24	0	A	HR	HR			
日寄存器	86H	87H	01～28,29,30,31	0	0	10DATE		DATE			
月寄存器	88H	89H	01～12	0	0	0	10M	MONTH			
星期寄存器	8AH	8BH	01～07	0	0	0	0	0	DAY		
年寄存器	8CH	8DH	00～99	10YEAR				YEAR			

DS1302 内部的 RAM 分为两类，一类是单个 RAM 单元，共 31 个，每个单元为一个 8 位的字节，其命令控制字为 COH～FDH，其中奇数为读操作，偶数为写操作；再一类为突发方式下的 RAM，此方式下可一次性读写所有的 RAM 的 31 个字节，命令控制字为 FEH（写）、FFH（读）。

我们现在已经知道了控制寄存器和 RAM 的逻辑地址，接着就需要知道如何通过外部接口来访问这些资源。单片机是通过简单的同步串行通信与 DS1302 通信的，每次通信都必须由单片机发起，无论是读还是写操作，单片机都必须先向 DS1302 写入一个命令帧，这个帧的格式见表 11-11，最高位 bit7 固定为 1，bit6 决定操作是针对 RAM 还是时钟寄存器，接着的 5bit 是 RAM 或时钟寄存器在 DS1302 的内部地址，最后一个 bit 表示这次操作是读操作或是写操作。

物理上，DS1302 的通信接口由 3 个口线组成，即 RST、SCLK、I/O。其中 RST 从低电平变成高电平启动一次数据传输过程，SCLK 是时钟线，I/O 是数据线。具体的读写时序如图 11-51 所示，但是请注意，无论是哪种同步通信类型的串行接口，都是对时钟信号敏感的，而且一般数据写入有效是在上升沿，读出有效是在下降沿（DS1302 正是如此的，但是在芯片手册里没有明确说明），如果不是特别确定，则把程序设计成这样：平时 SCLK 保持低电平，在时钟变动前设置数据，在时钟变动后读取数据，即数据操作总是在 SCLK 保持为低电平时，相邻的操作之间间隔有一个上升沿和一个下降沿。

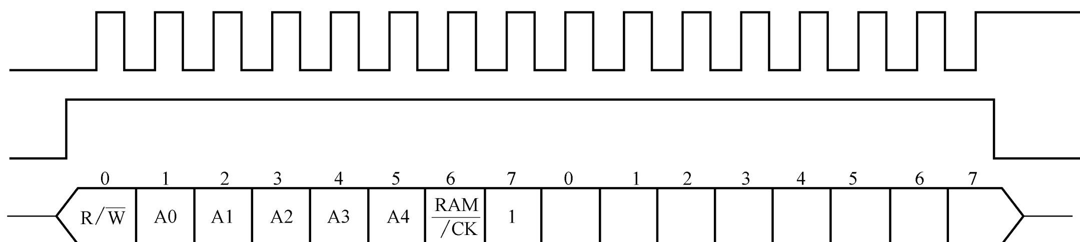


图 11-51　DS1302 的命令字结构

11.6.2　DS1302 的软硬件设计实例

本例将实现对 DS1302 的读写操作，将时钟数据在 1602LCD 上显示出来。为了便于程序实现，只对 DS1302 的时、分、秒进行读写。把 DS1302 的初始时间自定义为 09 年 08 月 08 日 12 时 30 分 00 秒。

1. 硬件原理图（见图 11-52）

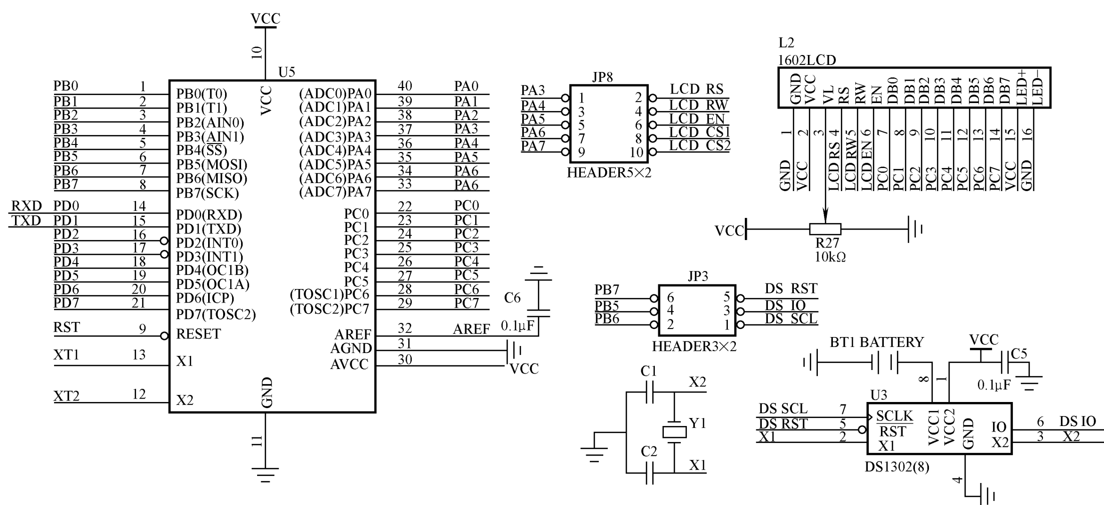


图 11-52　硬件原理图

2. 软件设计

```
/ ****************************************************************************** /
/ * DS1302 测试程序                                                             * /
/ * 目标器件:ATmega16                                                          * /
/ * 晶振:RC 1MHz                                                               * /
/ * 编译环境:ICCAVR 6.31A                                                      * /
/ ****************************************************************************** /

/ ***************************** 包含头文件 ***************************** /
#include  <iom16v.h>
#include  <macros.h>

/ ***************************** 端口定义 ***************************** /
#define    Enableon     PORTA |= BIT(5);       //EN 拉高
#define    Enableoff    PORTA &= ~BIT(5);      //EN 拉低
#define    RSon         PORTA |= BIT(3);       //RS 拉高
#define    RSoff        PORTA &= ~BIT(3);      //RS 拉低
#define    RWon         PORTA |= BIT(4);       //RW 拉高
#define    RWoff        PORTA &= ~BIT(4);      //RW 拉低

#define    DS1302_CLK_Low       PORTB &= ~BIT(6);
#define    DS1302_CLK_High      PORTB |= BIT(6);
#define    DS1302_IO_Low        PORTB &= ~BIT(5);
#define    DS1302_IO_High       PORTB |= BIT(5);
```

```
#define     DS1302_RST_High          PORTB |=BIT(7);
#define     DS1302_RST_Low           PORTB &= ~BIT(7);

#define     SomeNOP()                NOP();NOP();NOP();NOP();

/ ***************************** 全局变量 ***************************** /
unsigned char second,minute,hour,day,month,year;

/ ***********************************************************************
函数功能:延时程序
入口参数:ms
出口参数:
*********************************************************************** /
void delay(unsigned char ms)
{
    int i;
    while(ms--)
    {
      for(i = 0; i< 250; i++)
      {
        NOP();
        NOP();
        NOP();
        NOP();
      }
    }
}

/ ***********************************************************************
函数功能:等待 LCD 空闲程序
入口参数:
出口参数:
*********************************************************************** /
void LCD_wait(void)
{
    RSoff;//Dioff;
    RWon;
    DDRC =0x00;              //输入
    NOP();
    Enableon;
```

```
    while(PINC&0x80);
    Enableoff;
    DDRC =0xff;                //输出
}

/ ************************************************************************
函数功能:LCD 写指令使能程序
入口参数:cmd
出口参数:
 ************************************************************************ /
void write_command(unsigned char cmd)
{
    LCD_wait();
    RSoff;
    RWoff;
    Enableoff;
    NOP();
    Enableon;
    PORTC =cmd;
    Enableoff;
}

/ ************************************************************************
函数功能:设定显示位置程序
入口参数:pos
出口参数:
 ************************************************************************ /
void lcd_pos(unsigned char pos)
{
    write_command(pos | 0x80);
}

/ ************************************************************************
函数功能:写数据到 LCD 程序
入口参数:dat
出口参数:
 ************************************************************************ /
void write_data(unsigned char dat)
{
    LCD_wait();
```

```
    RSon;
    RWoff;
    Enableoff;
    NOP();
    Enableon;
    PORTC = dat;
    Enableoff;
}

/ *************************************************************************
函数功能:写入显示数字到 LCD
入口参数:
出口参数:
************************************************************************* /
void lcd_wno(unsigned char i)
{
    PORTC = i|0x30;
    RSon;
    RWoff;
    Enableon;
    NOP();
    //NOP();
    //NOP();
    Enableoff;
}

/ *************************************************************************
函数功能:LCD 初始化程序
入口参数:
出口参数:
************************************************************************* /
void lcd_init(void)
{
    delay(15);
    write_command(0x38);
    delay(100);
    write_command(0x01);
    write_command(0x06);
    write_command(0x0c);
}
```

```
/ ***********************************************************************
函数功能:向 DS1302 送 1 个字节数据程序
入口参数:byte1
出口参数:
*********************************************************************** /
void InputByte(unsigned char byte1)
{
    char count = 8;
    do
    {
        DS1302_CLK_Low;
        SomeNOP();
        if(byte1&0x01)
        {DS1302_IO_High;}
        else
        {DS1302_IO_Low;}
        SomeNOP();
        DS1302_CLK_High;
        SomeNOP();
        DS1302_CLK_Low;
        byte1 > > = 1;
      }while(--count);
}

/ ***********************************************************************
函数功能:读 DS1302 1 个字节程序
入口参数:
出口参数:data
*********************************************************************** /
unsigned char OutputByte(void)
{
        char count = 8;
        unsigned char data = 0;
        DDRB & = 0xdf;          //PB5 输入
        do
        {
            data > > = 1;
          DS1302_CLK_Low;
          SomeNOP();
```

```
            if(PINB&0x20)
            data| =0x80;
            DS1302_CLK_High;
            SomeNOP();
            DS1302_CLK_Low;
        }while(--count);
        DDRB =0xff;
        return(data);
}

/ *************************************************************************
函数功能:向 DS1302 某地址写 1 个字节数据程序
入口参数:addr,Dat
出口参数:
************************************************************************* /
void write_ds1302(unsigned char addr,unsigned char Dat)
{
    DS1302_RST_Low;
    SomeNOP();
    DS1302_CLK_Low;
    SomeNOP();
    DS1302_RST_High;
    InputByte(addr);
    InputByte(Dat);
    DS1302_CLK_High;
    SomeNOP();
    DS1302_RST_Low;
}

/ *************************************************************************
函数功能:读 DS1302 地址程序
入口参数:addr
出口参数:Dat
************************************************************************* /
unsigned char read_ds1302(unsigned char addr)
{
    unsigned char Dat;
    DS1302_RST_Low;
    SomeNOP();
    DS1302_CLK_Low;
```

```
    SomeNOP( );
    DS1302_RST_High;
    InputByte( addr);
    Dat = OutputByte( );
    DS1302_CLK_High;
    SomeNOP( );
    DS1302_RST_Low;
    return( Dat);
}

/ ************************************************************************
函数功能:初始化 DS1302 程序
入口参数:
出口参数:
************************************************************************ /
void initial_ds1302( )
{
    write_ds1302(0x8e,0x00);          //写保护寄存器,在对时钟或 RAM 写前 WP 一
                                        定要为 0
    write_ds1302(0x8c,0x09);          //09 年
    write_ds1302(0x88,0x08);          //08 月
    write_ds1302(0x86,0x29);          //08 日
    write_ds1302(0x84,0x21);          //12 时
    write_ds1302(0x82,0x32);          //30 分
    write_ds1302(0x80,0x10);          //00 秒
    write_ds1302(0x8e,0x80);          //写保护寄存器
}

/ ************************************************************************
函数功能:读 DS1302 时间程序
入口参数:
出口参数:second,minute,hour,day,month,year
************************************************************************ /
void read_time( )
{
    second = read_ds1302(0x81);          //秒寄存器
    minute = read_ds1302(0x83);          //分
    hour = read_ds1302(0x85);            //时
    day = read_ds1302(0x87);             //日
    month = read_ds1302(0x89);           //月
```

```
    year = read_ds1302(0x8d);               //年
}

/ ************************************************************************
函数功能:主程序
入口参数:
出口参数:
************************************************************************ /
void main(void)
{
    DDRA = 0xff;                            //1602LCD Contrl-(PA3 ~ PA5)
    DDRC = 0xff;                            //1602LCD Data  -(PC0 ~ PC7)
    DDRB = 0xff;                            //DS1302-SDA,SCL,RST

    lcd_init();                             //初始化 LCD
    delay(100);
    initial_ds1302();
    delay(100);
    while(1)
    {
    read_time();

    //hour
    lcd_pos(0x40);                          //显示位置
    lcd_wno(hour/16);                       //显示高位
    lcd_pos(0x41);                          //显示位置
    lcd_wno(hour%16);                       //显示低位
    //minute
    lcd_pos(0x43);                          //显示位置
    lcd_wno(minute/16);                     //显示高位
    lcd_pos(0x44);                          //显示位置
    lcd_wno(minute%16);                     //显示低位
    //second
    lcd_pos(0x46);                          //显示位置
    lcd_wno(second/16);                     //显示高位
    lcd_pos(0x47);                          //显示位置
    lcd_wno(second%16);                     //显示低位

    }

}
```

11.7　ADC 应用实例

在工业控制和智能化仪表中,通常由微型计算机进行实时控制及实时数据处理。计算机所加工的信息总是数字量,而被控制或被测量的有关参量往往是连续变化的模拟量,如温度、速度、压力等,与此对应的电信号是模拟信号。模拟量的存储和处理比较困难,不适合作为远距离传输且易受干扰。在一般的工业应用系统中,传感器把非电量的模拟信号变成与之对应的模拟信号,然后经模拟到数字转换电路将模拟信号转成对应的数字信号送微机处理。这就是一个完整的信号链,模拟到数字的转换过程就是我们经常接触到的 ADC 电路。ATmega16 片内自带 8 通道 10 位 ADC,可以方便地采集外部模拟电压。

11.7.1　ATmega16 片内 ADC 内部寄存器

ATmega16 内部 ADC 主要功能已在第 2 章中作了详细介绍,本节针对实际应用主要对内部寄存器进行再一次说明。

1. ADC 多工选择寄存器——ADMUX

bit	7	6	5	4	3	2	1	0
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

(1) bit 7:6-REFS1:0:参考电压选择

如表 11-13 所示，通过这几位可以选择参考电压。如果在转换过程中改变了它们的设置，只有等到当前转换结束（ADCSRA 寄存器的 ADIF 置位）之后改变才会起作用。如果在 AREF 引脚上施加了外部参考电压，内部参考电压就不能被选用了。

表 11-13　ADC 参考电压选择功能表

REFS1	REFS0	参考电压选择
0	0	AREF,内部 VREF 关
0	1	AVCC,AREF 引脚外加滤波电容
1	0	保留
1	1	2.56V 的片内基准电源,AREF 引脚外加滤波电容

(2) bit 5-ADLAR:ADC 转换结果左对齐

ADLAR 影响 ADC 转换结果在 ADC 数据寄存器中的存放形式。ADLAR 置位时转换结果为左对齐，否则为右对齐。ADLAR 的改变将立即影响 ADC 数据寄存器的内容，不论是否有转换正在进行。关于这一位的完整描述请见“ADC 数据寄存器——ADCL 和 ADCH”。

(3) bit 4:0-MUX4:0:模拟通道与增益选择位选择

如表 11-14 所示，如果在转换过程中改变这几位的值，那么只有到转换结束（ADCSRA 寄存器的 ADIF 置位）后新的设置才有效。

表 11-14　输入通道与增益表

MUX4:0	单端输入	正差分输入	负差分输入	增益
00000	ADC0			
00001	ADC1		N/A	
00010	ADC2			

（续）

MUX4:0	单端输入	正差分输入	负差分输入	增益
00011	ADC3	N/A		
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10X
01001		ADC1	ADC0	10X
01010		ADC0	ADC0	200X
01011		ADC1	ADC0	200X
01100		ADC2	ADC2	10X
01101		ADC3	ADC2	10X
01110		ADC2	ADC2	200X
01111		ADC3	ADC2	200X
10000		ADC0	ADC1	1X
10001		ADC1	ADC1	1X
10010		ADC2	ADC1	1X
10011		ADC3	ADC1	1X
10100		ADC4	ADC1	1X
10101		ADC5	ADC1	1X
10110		ADC6	ADC1	1X
10111		ADC7	ADC1	1X
11000		ADC0	ADC2	1X
11001		ADC1	ADC2	1X
11010		ADC2	ADC2	1X
11011		ADC3	ADC2	1X
11100		ADC4	ADC2	1X
11101		ADC5	ADC2	1X
11110	1.23V(V_{BG})	N/A		
11111	0V(GND)			

2. ADC 控制和状态寄存器 A——ADCSRA

bit	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

（1）bit 7-ADEN：ADC 使能

ADEN 置位即启动 ADC，否则 ADC 功能关闭。在转换过程中关闭 ADC 将立即中止正在进行的转换。

（2）bit 6-ADSC：ADC 开始转换

在单次转换模式下，ADSC 置位将启动一次 ADC 转换。在连续转换模式下，ADSC 置位将启动首次转换。第一次转换（在 ADC 启动之后置位 ADSC，或者在使能 ADC 的同时置位 ADSC）需要 25 个 ADC 时钟周期，而不是正常情况下的 13 个。第一次转换执行 ADC 初始化的工作。在转换进行过程中读取 ADSC 的返回值为“1”，直到转换结束。ADSC 清零不产生任何动作。

（3）bit 5-ADATE：ADC 自动触发使能

当该位写 1，将启动 ADC 自动触发使能。触发信号的上升沿启动 ADC 转换。触发信号

源通过 SFIOR 寄存器的 ADC 触发信号源选择位 ADTS 设置。

（4）bit 4-ADIF：ADC 中断标志

在 ADC 转换结束，且数据寄存器被更新后，ADIF 置位。如果 ADIE 及 SREG 中的全局中断使能位 I 也置位，ADC 转换结束中断服务程序即得以执行，同时 ADIF 硬件清零。此外，还可以通过向此标志写 1 来清 ADIF。要注意的是，如果对 ADCSRA 进行读—修改—写操作，那么待处理的中断会被禁止。这也适用于 SBI 及 CBI 指令。

（5）bit 3-ADIE：ADC 中断使能

若 ADIE 及 SREG 的全局中断使能位 I 置位，ADC 转换结束中断即被激活。

（6）bit 2:0-ADPS2:0：ADC 预分频器选择位

这几位确定了 XTAL 与 ADC 输入时钟之间的分频因子。ADC 预分频选择见表 11-15。

表 11-15　ADC 预分频选择表

ADPS2	ADPS1	ADPS0	分 频 因 子
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

3. ADC 数据寄存器——ADCL 和 ADCH

ADLAR = 0

bit	15	14	13	12	11	10	9	8	ADCH
							ADC9	ADC8	
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
读/写	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
初值	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

bit	15	14	13	12	11	10	9	8	ADCH
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	
	ADC1	ADC0							ADCL
	7	6	5	4	3	2	1	0	
读/写	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
初值	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

注：ADC9：表示 ADC 转换的结果。

ADC 转换结束后，转换结果存于这两个寄存器中。如果采用差分通道，结果用 2 的补码形式表示。

读取 ADCL 之后，ADC 数据寄存器一直要等到 ADCH 也被读出才可以进行数据更新。因此，如果转换结果为左对齐，且要求的精度不高于 8bit，那么仅需读取 ADCH 就足够了。

否则必须先读出 ADCL 再读 ADCH。

ADMUX 寄存器的 ADLAR 及 MUXn 会影响转换结果在数据寄存器中的表示方式。如果 ADLAR 为 1，那么结果为左对齐；反之（系统默认设置），结果为右对齐。

11.7.2　ADC 软硬件设计实例

通过对片内 ADC 模块的学习，将采集到的模块电压显示在数码管上。为了便于说明学习，我们暂时采集单个通道的信号。参考电压源选择将通过内部寄存器设置成 AVCC，外部模拟电压通过电位器分压得到，范围为 0 ~5V。

1. 硬件原理图（见图 11-53）

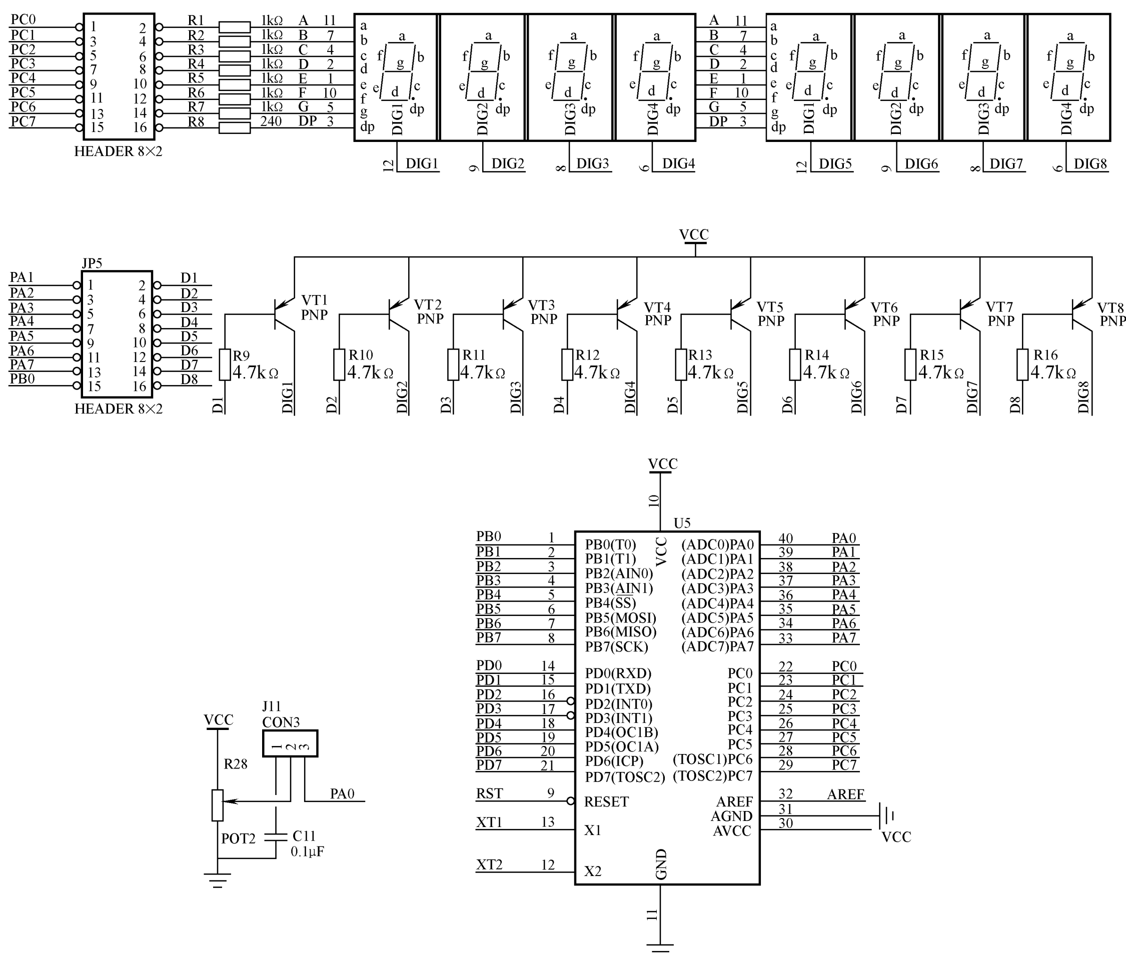


图 11-53　ADC 实验原理图

2. 软件设计

```
/ ************************************************************************ /
/ * ADC 测试程序                                                           * /
/ * 目标器件:ATmega16                                                      * /
/ * 晶振:RC 1MHz                                                           * /
/ * 编译环境:ICCAVR 6.31A                                                   * /
```

```
/ ******************************************************************* /

/ ************************* 包含头文件 ************************* /
#include  < iom16v. h >
#include  < macros. h >

/ ************************* 端口定义 ************************* /

/ ************************* 数码管段码表 ************************* /
extern const  unsigned  char  tab[ ] = { 0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
0x80,0x90,
            /*    0    1    2    3    4    5    6    7    8    9   */
                  0x88,0x83,0xC6,0xA1,0x86,0x8E};
            /*    a    b    c  d    e    f   */


/ ************************* 全局变量 ************************* /
unsigned int adc_rel;
int dat;


/ *******************************************************************
函数功能:数码管延时程序
入口参数:
出口参数:
 ******************************************************************* /
void delay_1us(void)                  //1μs 延时函数
  {
  asm("nop");
  }

void delay_nus(unsigned int n)        //N μs 延时函数
  {
  unsigned int i =0;
  for(i =0;i < n;i + + )
  delay_1us();
  }

void delay_1ms(void)                  //1ms 延时函数
  {
```

```
  unsigned int i;
  for(i=0;i<140;i++);
  }

void delay_nms(unsigned int n)        //N ms 延时函数
  {
  unsigned int i=0;
  for(i=0;i<n;i++)
  delay_1ms();
  }

/************************************************************************
函数功能:显示子程序
入口参数:k
出口参数:
************************************************************************/
void display(unsigned int k)
{
    PORTC=tab[k/1000];
    PORTA=0xef;
    delay_nms(2);

    PORTC=tab[k/100%10];
    PORTA=0xf7;
    delay_nms(2);

    PORTC=tab[k/10%10];
    PORTA=0xfb;
    delay_nms(2);

    PORTC=tab[k%10];
    PORTA=0xfd;
    delay_nms(2);
}


/************************************************************************
函数功能:ADC 初始化函数
入口参数:
出口参数:
```

```
*********************************************************************/
void adc_init(void)
{
  ADCSRA = 0x00;                 //关 ADC
  ACSR = (1 << ACD);             //关闭模拟比较器
ADMUX  = (1 << REFS0);//AVCC
ADCSRA  = (1 << ADEN)|(1 << ADSC)|(1 << ADATE)|(1 << ADIF)|(1 << ADIE)
|(1 << ADPS2)|(1 << ADPS1);//64 分频
SFIOR & = 0x1f;
}

/*********************************************************************
函数功能:ADC 中断函数
入口参数:
出口参数:
*********************************************************************/
#pragma interrupt_handler adc_isr:15
void adc_isr(void)
{
    ADCSRA  =  0x00;
    adc_rel = ADCL;
    adc_rel| = (unsigned int)(ADCH << 8);
    ADCSRA  = (1 << ADEN)|(1 << ADSC)|(1 << ADIE);
}


/*********************************************************************
函数功能:主程序
入口参数:
出口参数:
*********************************************************************/
void main(void)
{
    DDRC = 0xff;
    PORTC = 0xff;
    DDRA = 0xfe;
    PORTA = 0xfe;

    adc_init();
    SEI();
```

```
    display(0);
    while(1)
    {
      display(adc_rel);
    }
}
```

11.8　1602 字符型 LCD 应用实例

在日常生活中，我们对液晶显示器并不陌生。液晶显示模块已作为很多电子产品的通用器件，如在计算器、万用表、电子表及很多家用电子产品中都可以看到，显示的主要是数字、专用符号和图形。在单片机的人机交流界面中，一般的输出显示方式有以下几种：发光管、LED 数码管、液晶显示器。发光管和 LED 数码管比较常用，软硬件都比较简单，在前面章节已经介绍过，在此不作介绍，本章重点介绍液晶显示器的应用。在单片机系统中应用液晶显示器作为输出器件有以下几个优点：

（1）显示质量高

由于液晶显示器每一个点在收到信号后就一直保持那种色彩和亮度，恒定发光，而不像阴极射线管显示器（CRT）那样需要不断刷新新亮点。因此，液晶显示器画质高且不会闪烁。

（2）数字式接口

液晶显示器都是数字式的，与单片机系统的接口更加简单可靠，操作更加方便。

（3）体积小、重量轻

液晶显示器通过显示屏上的电极控制液晶分子状态来达到显示的目的，在重量上比相同显示面积的传统显示器要轻得多。

（4）功耗低

相对而言，液晶显示器的功耗主要消耗在其内部的电极和驱动 IC 上，因而耗电量比其他显示器要少得多。

11.8.1　液晶显示简介

1. 液晶显示原理

液晶显示的原理是利用液晶的物理特性，通过电压对其显示区域进行控制，有电就有显示，这样即可以显示出图形。液晶显示器具有厚度薄、适用于大规模集成电路直接驱动、易于实现全彩色显示的特点，目前已经被广泛应用在笔记本电脑、数字摄像机、PDA 移动通信工具等众多领域。

2. 液晶显示器的分类

液晶显示的分类方法有很多种，通常可按其显示方式分为段式、字符式、点阵式等。除了黑白显示外，液晶显示器还有多灰度有彩色显示等。如果根据驱动方式来分，可以分为静态驱动（Static）、单纯矩阵驱动（Simple Matrix）和主动矩阵驱动（Active Matrix）3 种。

3. 液晶显示器各种图形的显示原理

（1）线段的显示

点阵图形式液晶由 $M \times N$ 个显示单元组成，假设 LCD 显示屏有 64 行，每行有 128 列，每 8 列对应 1B 的 8 位，即每行由 16B，共 16×8 = 128 个点组成，屏上 64×16 个显示单元与显示 RAM 区 1024B 相对应，每一字节的内容和显示屏上相应位置的亮暗对应。例如屏的第 1 行的亮暗由 RAM 区的 000H ~ 00FH 的 16B 的内容决定，当（000H）= FFH 时，则屏幕的左上角显示一条短亮线，长度为 8 个点；当（3FFH）= FFH 时，则屏幕的右下角显示一条短亮线；当（000H）= FFH，（001H）= 00H，（002H）= 00H……（00EH）= 00H，（00FH）= 00H 时，则在屏幕的顶部显示一条由 8 段亮线和 8 条暗线组成的虚线。这就是 LCD 显示的基本原理。

（2）字符的显示

用 LCD 显示一个字符时比较复杂，因为一个字符由 6×8 或 8×8 点阵组成，既要找到和显示屏幕上某几个位置对应的显示 RAM 区的 8 字节，还要使每个字节的不同位为“1”，其他的为“0”，为“1”的点亮，为“0”的不亮。这样一来就组成某个字符。但对于内带字符发生器的控制器来说，显示字符就比较简单了，可以让控制器工作在文本方式，根据在 LCD 上开始显示的行列号及每行的列数找出显示 RAM 对应的地址，设立光标，在此送上该字符对应的代码即可。

（3）汉字的显示

汉字的显示一般采用图形的方式，事先从微机中提取要显示的汉字的点阵码（一般用字模提取软件），每个汉字占 32B，分左右两半，各占 16B，左边为 1、3、5……右边为 2、4、6……根据在 LCD 上开始显示的行列号及每行的列数可找出显示 RAM 对应的地址，设立光标，送上要显示的汉字的第 1 个字节，光标位置加 1，送第 2 个字节，换行按列对齐，送第 3 个字节……直到 32B 显示完就可以 LCD 上得到一个完整汉字。

11.8.2　1602 字符型 LCD 简介

字符型液晶显示模块是一种专门用于显示字母、数字、符号等点阵式 LCD，目前常用 16×1、16×2、20×2 和 40×2 行等的模块。下面以市场上常见的 1602 字符型液晶显示器为例，介绍其用法。一般 1602 字符型液晶显示器实物如图 11-54 所示。

1. 1602LCD 的基本参数及引脚功能

1602LCD 分为带背光和不带背光两种，其控制器大部分为 HD44780，带背光的比不带背光的厚，是否带背光在应用中并无差别，两者尺寸差别如图 11-55 所示。

（1）1602LCD 主要技术参数

显示容量：　16×2 个字符

芯片工作电压：　4.5 ~ 5.5V

工作电流：　2.0mA（5.0V）

模块最佳工作电压：5.0V

字符尺寸（$W \times H$）：2.95mm×4.35mm

（2）引脚功能说明

1602LCD 采用标准的 14 脚（无背光）或 16 脚（带背光）接口，各引脚接口说明见表 11-16。

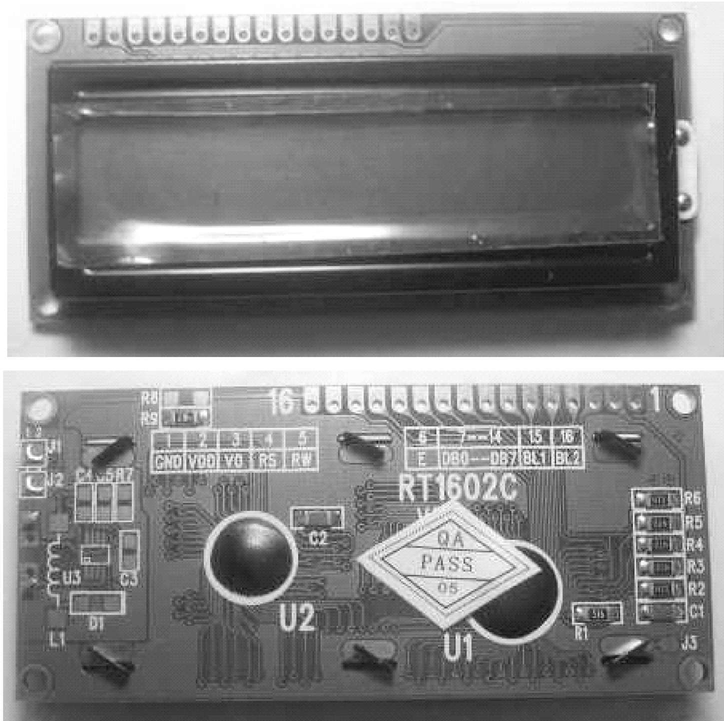


图 11-54　1602 字符型液晶显示器实物图

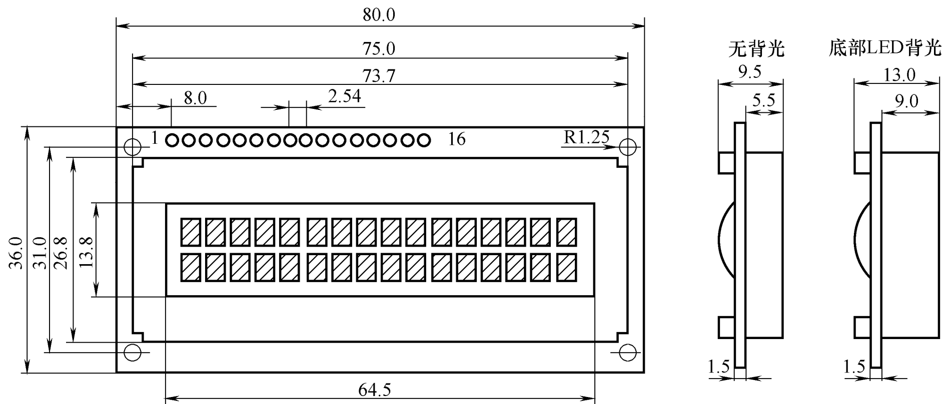


图 11-55　1602LCD 尺寸图

表 11-16　引脚接口说明表

编号	符号	引 脚 说 明	编号	符号	引 脚 说 明
1	VSS	电源地	9	D2	数据
2	VDD	电源正极	10	D3	数据
3	VL	液晶显示偏压	11	D4	数据
4	RS	数据/命令选择	12	D5	数据
5	R/W	读/写选择	13	D6	数据
6	E	使能信号	14	D7	数据
7	D0	数据	15	BLA	背光源正极
8	D1	数据	16	BLK	背光源负极

第 1 脚：VSS 为地电源。

第 2 脚：VDD 接 5V 正电源。

第 3 脚：VL 为液晶显示器对比度调整端，接正电源时对比度最弱，接地时对比度最高，对比度过高时会产生“鬼影”，使用时可以通过一个 10kΩ 的电位器调整对比度。

第 4 脚：RS 为寄存器选择，高电平时选择数据寄存器，低电平时选择指令寄存器。

第 5 脚：R/W 为读写信号线，高电平时进行读操作，低电平时进行写操作。当 RS 和 R/W 共同为低电平时可以写入指令或者显示地址，当 RS 为低电平、R/W 为高电平时可以读忙信号，当 RS 为高电平、R/W 为低电平时可以写入数据。

第 6 脚：E 端为使能端，当 E 端由高电平跳变成低电平时，液晶模块执行命令。

第 7 ~ 14 脚：D0 ~ D7 为 8 位双向数据线。

第 15 脚：背光源正极。

第 16 脚：背光源负极。

2. 1602LCD 的指令说明及时序

1602 液晶模块内部的控制器共有 11 条控制指令，见表 11-17。

表 11-17　控制命令表

序号	指　　令	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	清显示	0	0	0	0	0	0	0	0	0	1
2	光标复位	0	0	0	0	0	0	0	0	1	*
3	置输入模式	0	0	0	0	0	0	0	1	I/D	S
4	显示开/关控制	0	0	0	0	0	0	1	D	C	B
5	光标或字符移位	0	0	0	0	0	1	S/C	R/L	*	*
6	置功能	0	0	0	0	1	DL	N	F	*	*
7	置字符发生存储器地址	0	0	0	1	字符发生存储器地址					
8	置数据存储器地址	0	0	1	显示数据存储器地址						
9	读忙标志或地址	0	1	BF	计数器地址						
10	写数到 CGRAM 或 DDRAM	1	0	要写的数据内容							
11	从 CGRAM 或 DDRAM 读数	1	1	读出的数据内容							

1602 液晶模块的读写操作、屏幕和光标的操作都是通过指令编程来实现的（说明：1 为高电平，0 为低电平）。

指令 1：清显示，指令码 01H，光标复位到地址 00H 位置。

指令 2：光标复位，光标返回到地址 00H。

指令 3：光标和显示模式设置，I/D：光标移动方向，高电平右移，低电平左移；S：屏幕上所有文字是否左移或者右移，高电平表示有效，低电平则无效。

指令 4：显示开/关控制。D：控制整体显示的开与关，高电平表示开显示，低电平表示关显示；C：控制光标的开与关，高电平表示有光标，低电平表示无光标；B：控制光标是否闪烁，高电平闪烁，低电平不闪烁。

指令 5：光标或显示移位，S/C：高电平时移动显示的文字，低电平时移动光标。

指令 6：功能设置命令，DL：高电平时为 4 位总线，低电平时为 8 位总线；N：低电平时为单行显示，高电平时双行显示；F：低电平时显示 5 × 7 的点阵字符，高电平时显示 5 × 10 的点阵字符。

指令 7：字符发生器 RAM 地址设置。

指令 8：DDRAM 地址设置。

指令 9：读忙信号和光标地址，BF：为忙标志位，高电平表示忙，此时模块不能接收命令或者数据，如果为低电平表示不忙。

指令 10：写数据。

指令 11：读数据。

与 HD44780 相兼容的芯片时序表见表 11-18

表 11-18　基本操作时序表

读状态	输入	RS = L, R/W = H, E = H	输出	D0—D7 = 状态字
写指令	输入	RS = L, R/W = L, D0 ~ D7 = 指令码, E = 高脉冲	输出	无
读数据	输入	RS = H, R/W = H, E = H	输出	D0—D7 = 数据
写数据	输入	RS = H, R/W = L, D0 ~ D7 = 数据, E = 高脉冲	输出	无

读写操作时序如图 11-56 和图 11-57 所示。

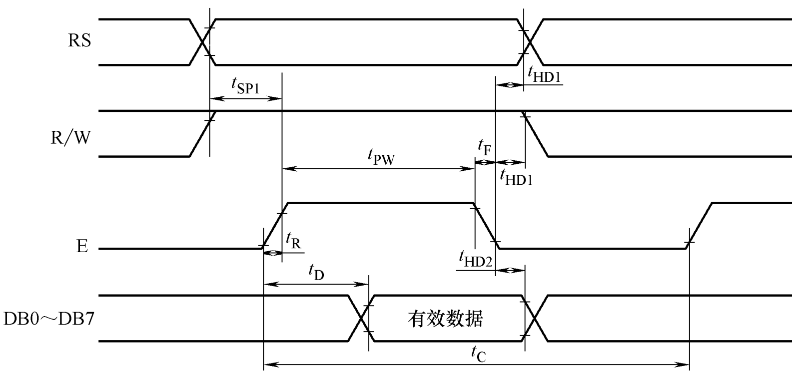


图 11-56　读操作时序

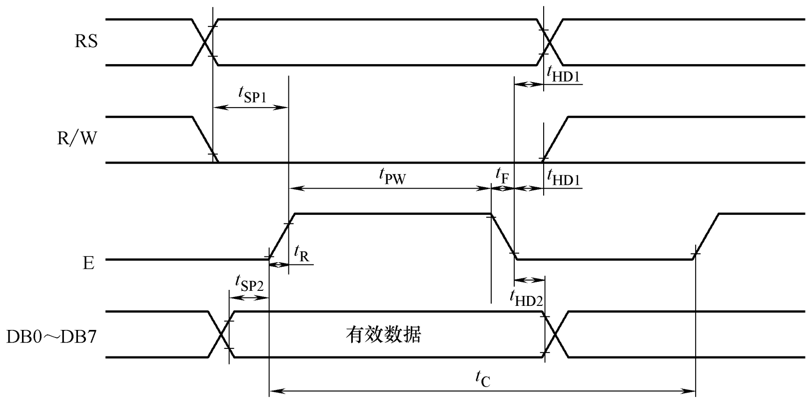


图 11-57　写操作时序

3. 1602LCD 的 RAM 地址映射及标准字库表

液晶显示模块是一个慢显示器件，所以在执行每条指令之前一定要确认模块的忙标志为低电平，表示不忙，否则此指令失效。要显示字符时要先输入显示字符地址，也就是告诉模块在哪里显示字符，图 11-58 是 1602 LCD 的内部显示地址。

例如，第 2 行第 1 个字符的地址是 40H，那么是否直接写入 40H 就可以将光标定位在第 2 行第 1 个字符的位置呢？这样不行，因为写入显示地址时要求最高位 D7 恒定为高电平 1，

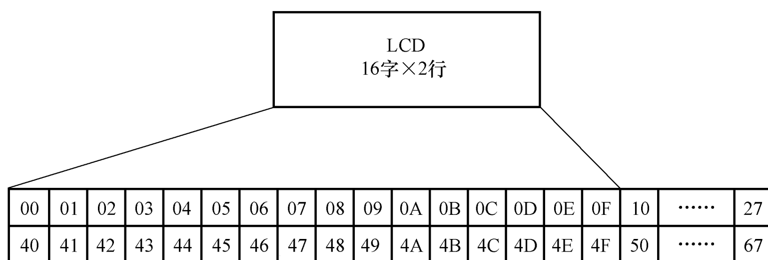


图 11-58　1602LCD 内部显示地址

所以实际写入的数据应该是 01000000B（40H）+ 10000000B（80H）= 11000000B（C0H）。

在对液晶模块的初始化中要先设置其显示模式，在液晶模块显示字符时光标是自动右移的，无需人工干预。每次输入指令前都要判断液晶模块是否处于忙的状态。

1602 液晶模块内部的字符发生存储器（CGROM）已经存储了 160 个不同的点阵字符图形，如图 11-59 所示，这些字符有：阿拉伯数字、英文字母的大小写、常用的符号和日文假名等，每一个字符都有一个固定的代码，比如大写的英文字母“A”的代码是 01000001B（41H），显示时模块把地址 41H 中的点阵字符图形显示出来，我们就能看到字母“A”

低位＼高位	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
××××0000	CGRAM (1)		0	∂	P	\	p		一	タ	三	α	P
××××0001	(2)	!	1	A	Q	a	q	ロ	ア	チ	ム	ä	q
××××0010	(3)	〃	2	B	R	b	r	┌	イ	川	メ	β	θ
××××0011	(4)	#	3	C	S	c	s	┘	ウ	ラ	モ	ε	∞
××××0100	(5)	$	4	D	T	d	t	\	エ	ト	セ	μ	Ω
××××0101	(6)	%	5	E	U	e	u	ロ	オ	ナ	ユ	B	0
××××0110	(7)	&	6	F	V	f	v	テ	カ	ニ	ヨ	P	Σ
××××0111	(8)	>	7	G	W	g	w	ア	キ	ヌ	ラ	g	π
××××1000	(1)	(	8	H	X	h	x	イ	ク	ネ	リ	∫	$\bar{X}$
××××1001	(2)	)	9	I	Y	i	y	ウ	ケ	┘	ル	−1	y
××××1010	(3)	*	:	J	Z	j	z	エ	コ	リ	レ	j	千
××××1011	(4)	+	:	K	[	k	{	オ	サ	ヒ	ロ	x	万
××××1100	(5)	フ	<	L	¥	1	|	セ	シ	フ	ワ	e	円
××××1101	(6)	–	=	M	]	m	}	ユ	エ	ヘ	ソ	≠	+
××××1110	(7)	•	>	N	^	n	-	ヨ	セ	ホ	ハ	$\bar{n}$	
××××1111	(8)	/	?	O	–	o	←	ツ	ソ	マ	ロ	ö	▒

图 11-59　字符代码与图形对应图

4. 1602LCD 的一般初始化（复位）过程

延时 15ms

写指令 38H（不检测忙信号）

延时 5ms

写指令 38H（不检测忙信号）

延时 5ms

写指令 38H（不检测忙信号）

以后每次写指令、读/写数据操作均需要检测忙信号

写指令 38H：显示模式设置

写指令 08H：显示关闭

写指令 01H：显示清屏

写指令 06H：显示光标移动设置

写指令 0CH：显示开及光标设置

11.8.3　1602LCD 的软硬件设计实例

通过以上介绍，我们已经可以对字符型显示器 1602LCD 进行软硬件开发。以实验板为硬件平台，在 LCD 上显示一个网址和电子邮件信息，具体为：在第 1 行显示 www. hificat. com，在第 2 行显示 98dian@ 163. com。

1. 硬件原理图

1602 液晶显示模块可以和单片机 ATmega128 直接接口，电路如图 11-60 所示。

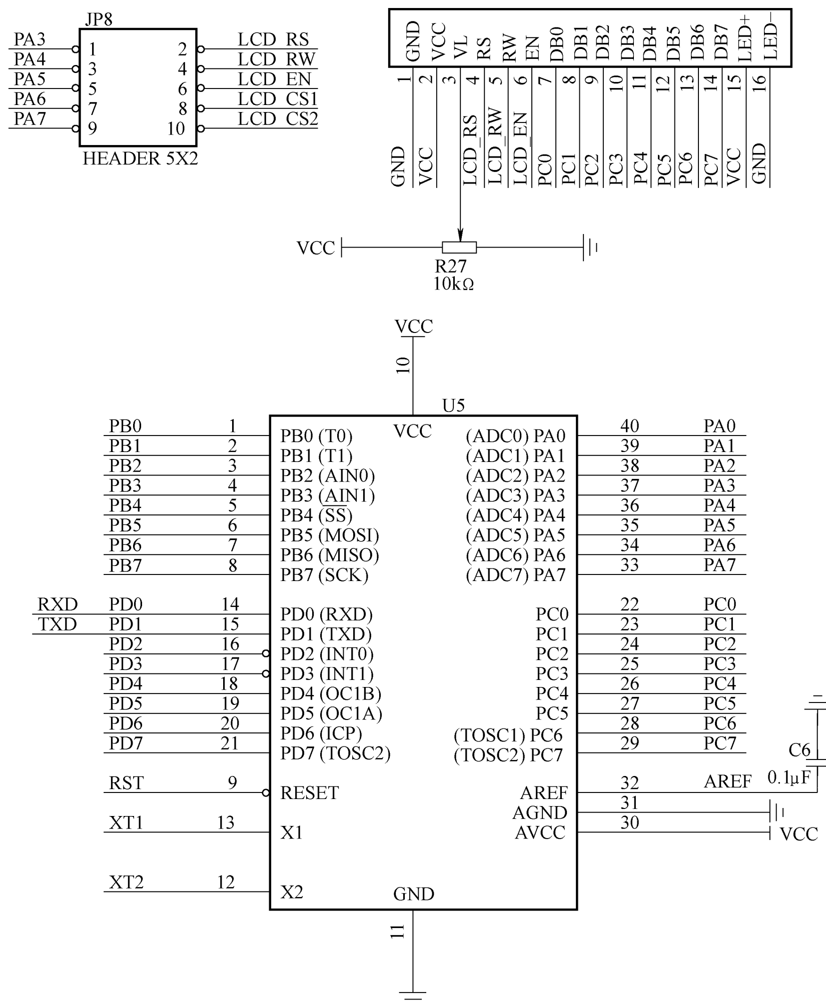


图 11-60　硬件原理图

2. 软件设计

```
/ ************************************************************************ /
/ * 1602LCD 测试程序                                                     * /
/ * 目标器件:ATmega16                                                    * /
/ * 晶振:RC 1MHz                                                         * /
/ * 编译环境:ICCAVR 6.31A                                                * /
/ ************************************************************************ /

/ ***************************** 包含头文件 ***************************** /
#include  <iom16v.h>
#include  <macros.h>

/ ***************************** 端口定义 ***************************** /
#define    Enableon     PORTA |= BIT(5);          //EN 拉高
#define    Enableoff    PORTA &= ~BIT(5);         //EN 拉低
#define    RSon         PORTA |= BIT(3);          //RS 拉高
#define    RSoff        PORTA &= ~BIT(3);         //RS 拉低
#define    RWon         PORTA |= BIT(4);          //RW 拉高
#define    RWoff        PORTA &= ~BIT(4);         //RW 拉低

/ ***************************** 字符定义 ***************************** /
extern const unsigned char dis1[ ]  = {"www.hificat.com"};   //显示字符串
extern const unsigned char dis2[ ]  = {"98dian@163.com"};

/ ************************************************************************
函数功能:延时程序
入口参数:ms
出口参数:
************************************************************************ /
void delay(unsigned char ms)                               //延时子程序
{
    int i;
    while(ms--)
    {
        for(i = 0; i < 250; i++)
        {
          NOP();
          NOP();
          NOP();
          NOP();
```

```
            }
        }
}

/ ************************************************************************
函数功能:LCD 延时等待程序
入口参数:
出口参数:
************************************************************************ /
void LCD_wait(void)                                  //等待 LCD 空闲
{
    RSoff;//Dioff;
    RWon;
    DDRC =0x00;                                      //输入
    NOP();
    Enableon;
    while(PINC&0x80);
    Enableoff;
    DDRC =0xff;                                      //输出
}

/ ************************************************************************
函数功能:LCD 写指令使能程序
入口参数:cmd
出口参数:
************************************************************************ /
void write_command(unsigned char cmd)
{
    LCD_wait();
    RSoff;
    RWoff;
    Enableoff;
    NOP();
    Enableon;
    PORTC = cmd;
    Enableoff;
}

/ ************************************************************************
函数功能:设定显示位置程序
```

```
入口参数:pos
出口参数:
************************************************************************/
void lcd_pos(unsigned char pos)
{
    write_command(pos | 0x80);
}

/************************************************************************
函数功能:LCD 写数据程序
入口参数:dat
出口参数:
************************************************************************/
void write_data(unsigned char dat)
{
    LCD_wait();
    RSon;
    RWoff;
    Enableoff;
    NOP();
    Enableon;
    PORTC = dat;
    Enableoff;
}

/************************************************************************
函数功能:LCD 初始化程序
入口参数:
出口参数:
************************************************************************/
void lcd_init(void)
{
    delay(15);
    write_command(0x38);
    delay(100);
    write_command(0x01);
    write_command(0x06);
    write_command(0x0c);
}
```

```
/ *******************************************************************
函数功能:主程序
入口参数:
出口参数:
 ******************************************************************* /
void main( void)
{
    char i;
    DDRA =0xff;
    DDRC =0xff;
    lcd_init( ) ;                                   //初始化 LCD
    delay(100) ;

    while(1)
    {
    lcd_pos(1) ;                                    //设置显示位置
    i  =  0;
    while( dis1[ i] !  ='\0')
    {
        write_data( dis1[ i]) ;                     //显示字符
        i + + ;
    }
    lcd_pos(0x41) ;                                 //设置显示位置
    i =0;
    while( dis2[ i] !  ='\0')
    {
        write_data( dis2[ i]) ;                     //显示字符
        i + + ;
    }
    }

}
```

11.9　12864 点阵型 LCD 应用实例

前文介绍了字符型 LCD 的基本应用，通过上一节的学习后，可以实现数字和字母的显示，但在现场应用中除了数字和字母外还要显示汉字。因为字符型 LCD 无法将汉字显示出来，所以要在显示汉字的场合一般都用点阵型 LCD。目前常用的点阵型 LCD 有 122 ×32、128 ×64、240 ×320 等。本节重点介绍 128 ×64 点阵显示屏的基本应用，128 ×64 点阵显示屏有 3 种控制器，分别是 KS0107 （KS0108）、T6963C 和 ST7920。3 种控制器主要区别是：

KS0107（KS0108）不带任何字库，T6963C 带 ASCII 码，ST7920 带国标二级字库（8000 多个汉字）。本节以不带字库的 KS0107（KS0108）控制器 LCD 为例，介绍汉字的基本显示方法。

11.9.1　点阵 LCD 的显示原理

在上节中我们已经了解了 LCD 基本的显示原理，字符型和点阵型 LCD 的主要区别是：字符型 LCD 只能显示数字和字母符号，所需存储空间有限，所以一般都把基本字库表固化在自带的 ROM 里；而不带汉字库的点阵显示屏则不同，每个字符和汉字都要用户自己取模。下面简单介绍一下显示字符和显示汉字的区别：

在数字电路中，所有的数据都是以 0 和 1 保存的，对 LCD 控制器进行不同的数据操作，可以得到不同的结果。对于显示英文操作，由于英文字母种类很少，只需要 8 位（1B）即可。而对于中文，常用的却有 6000 个以上，于是我们的 DOS 前辈想了一个办法，就是将 ASCⅡ表的高 128 位很少用到的数值以两个为一组来表示汉字，即汉字的内码。而剩下的低 128 位则留给英文字符使用，即英文的内码。

那么，得到了汉字的内码后，还仅是一组数字，那又如何在屏幕上去显示呢？这就涉及文字的字模，字模虽然也是一组数字，但它的意义却与数字的意义有了根本的变化，它是用数字的各位信息来记载英文或汉字的形状，如英文的“A”在字模的记载方式如图 11-61 所示。

英文字模　位代码　字模信息

0 0 0 0 0 0 0 0　0x00
0 0 0 0 0 0 0 0　0x00
0 0 0 1 0 0 0 0　0x10
0 0 1 1 1 0 0 0　0x38
0 1 1 0 1 1 0 0　0x6c
1 1 0 0 0 1 1 0　0xc6
1 1 0 0 0 1 1 0　0xc6
1 1 1 1 1 1 1 0　0xfe
1 1 0 0 0 1 1 0　0xc6
1 1 0 0 0 1 1 0　0xc6
1 1 0 0 0 1 1 0　0xc6
1 1 0 0 0 1 1 0　0xc6
0 0 0 0 0 0 0 0　0x00
0 0 0 0 0 0 0 0　0x00
0 0 0 0 0 0 0 0　0x00
0 0 0 0 0 0 0 0　0x00

图 11-61　“A”字模图

而中文的“你”在字模中的记载如图 11-62 所示。

中文字模　位代码　字模信息

0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0　0x08,0x80
0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0　0x08,0x80
0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0　0x08,0x80
0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 0　0x11,0xfe
0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0　0x11,0x02
0 0 1 1 0 0 1 0 0 0 0 0 0 1 0 0　0x32,0x04
0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 0　0x54,0x20
0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0　0x10,0x20
0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0　0x10,0xa8
0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0　0x10,0xa4
0 0 0 1 0 0 0 1 0 0 1 0 0 1 1 0　0x11,0x26
0 0 0 1 0 0 1 0 0 0 1 0 0 0 1 0　0x12,0x22
0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0　0x10,0x20
0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0　0x10,0x20
0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0　0x10,0xa0
0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0　0x10,0x40

图 11-62　“你”字模图

11.9.2　12864 点阵型 LCD 简介

12864 是一种图形点阵液晶显示器，它主要由行驱动器、列驱动器及 128×64 全点阵液

晶显示器组成。可完成图形显示，也可以显示 8×4 个（16×16 点阵）汉字。

主要技术参数和性能如下：

1）电源：VDD +5V；模块内自带 -10V 负压，用于 LCD 的驱动电压。

2）显示内容：128（列）×64（行）点。

3）全屏幕点阵。

4）7 种指令。

5）与 CPU 接口采用 8 位数据总线并行输入输出和 8 条控制线。

6）占空比 1/64。

7）工作温度：-20 ~ +60℃，存储温度：-30 ~ +70℃。

1. 12864LCD 的外形结构及引脚说明

其外形尺寸如图 11-63 所示。引脚说明见表 11-19。

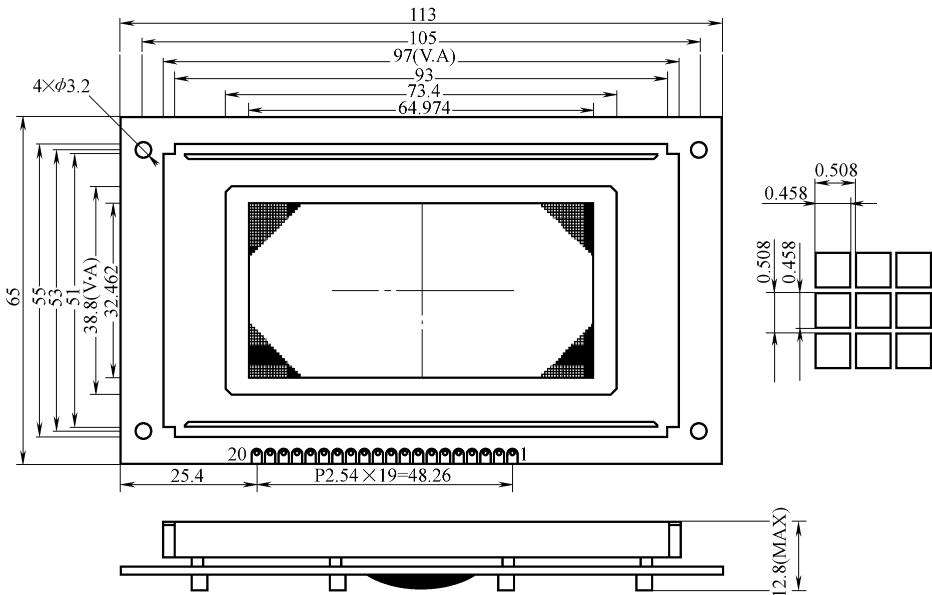


图 11-63　外形尺寸图

表 11-19　12864LCD 的引脚说明

引脚号	引脚名称	LEVER	引脚功能描述
1	VSS	0	电源地
2	VDD	+5.0V	电源电压
3	V0	—	液晶显示器驱动电压
4	D/I(RS)	H/L	D/I =“H”,表示 DB7 ~ DB0 为显示数据 D/I =“L”,表示 DB7 ~ DB0 为显示指令数据
5	R/W	H/L	R/W =“H”,E =“H”数据被读到 DB7 ~ DB0 R/W =“L”,E =“H→L”数据被写到 IR 或 DR
6	E	H/L	R/W =“L”,E 信号下降沿锁存 DB7 ~ DB0 R/W =“H”,E =“H”DDRAM 数据读到 DB7 ~ DB0
7	DB0	H/L	数据线
8	DB1	H/L	数据线
9	DB2	H/L	数据线
10	DB3	H/L	数据线

（续）

引脚号	引脚名称	LEVER	引脚功能描述
11	DB4	H/L	数据线
12	DB5	H/L	数据线
13	DB6	H/L	数据线
14	DB7	H/L	数据线
15	CS1	H/L	H：选择芯片（右半屏）信号
16	CS2	H/L	H：选择芯片（左半屏）信号
17	RET	H/L	复位信号，低电平复位
18	VOUT	-10V	LCD 驱动负电压
19	LED +	—	LED 背光板电源
20	LED -	—	LED 背光板电源

2. 12864LCD 的内部模块结构

此处介绍的液晶显示模块均是使用 KS0108B 及其兼容控制驱动器（例如 HD61202）作为列驱动器，同时使用 KS0107B 及其兼容驱动器（例如 HD61203）作为行驱动器的液晶模块。由于 KS0107B（或 HD61203）不与 MPU 发生联系，只要提供电源就能产生行驱动信号和各种同步信号，比较简单，在此就不作介绍。图 11-64 是 12864 的内部逻辑电路图，12864 共有两片 KS0108B 或兼容控制驱动器和一片 HD61203 或兼容驱动器。

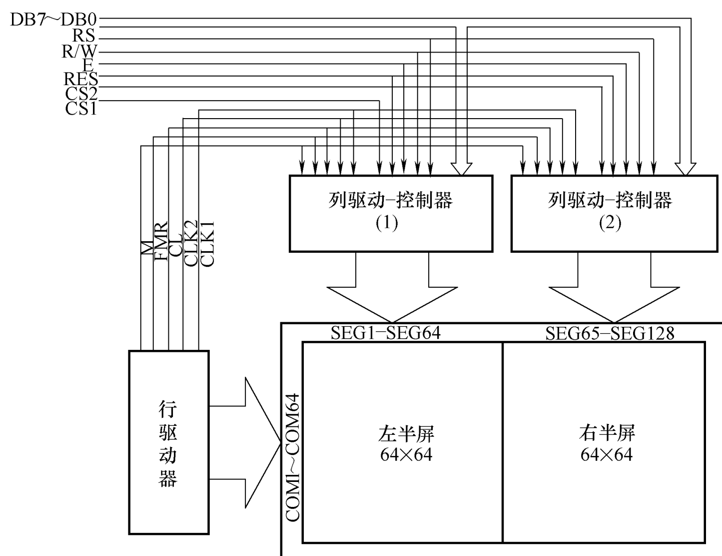


图 11-64　12864 的内部逻辑电路图

从图 11-64 可以看出，12864 的 LCD 基本由数据线（DB0～DB7）和控制线组成，左右半屏的显示控制由 CS1、CS2 控制，CS1 和 CS2 不同的组合可以控制屏幕左右显示。12864LCD 内部结构如图 11-65 所示。

在使用 12864LCD 前先必须了解以下功能器件才能进行编程。12864 内部功能器件及相关功能如下：

（1）指令寄存器（IR）

IR 是用于寄存指令码，与数据寄存器数据相对应。当 D/I＝0 时，在 E 信号下降沿的作用下，指令码写入 IR。

（2）数据寄存器（DR）

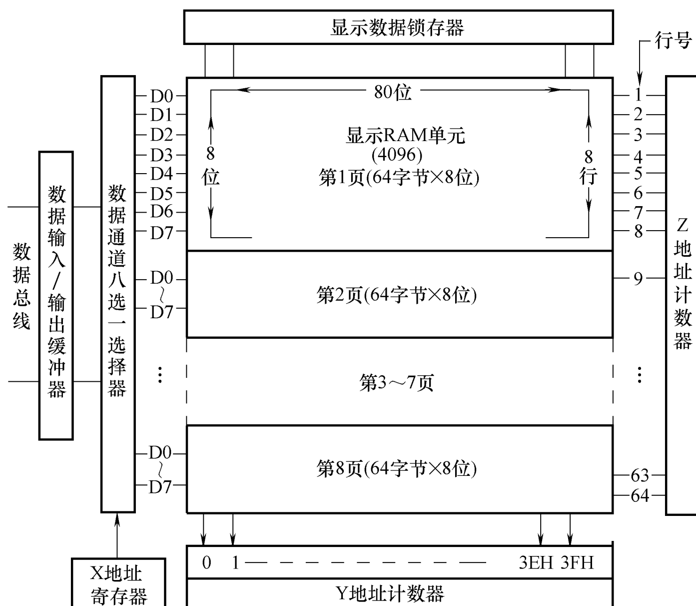


图 11-65　内部结构

DR 是用于寄存数据的，与指令寄存器寄存指令相对应。当 D/I = 1 时，在下降沿作用下，图形显示数据写入 DR，或在 E 信号高电平作用下由 DR 读到 DB7 ~ DB0 数据总线。DR 和 DDRAM 之间的数据传输是模块内部自动执行的。

（3）忙标志（BF）

BF 标志提供内部工作情况。BF = 1 表示模块在内部操作，此时模块不接受外部指令和数据。BF = 0 时，模块为准备状态，随时可接收外部指令和数据。

利用 STATUS READ 指令，可以将 BF 读到 DB7 总线，从而检验模块的工作状态。

（4）显示控制触发器（DFF）

此触发器是用于模块屏幕显示开和关的控制。DFF = 1 为开显示（DISPLAY OFF），DDRAM 的内容就显示在屏幕上，DFF = 0 为关显示（DISPLAY OFF）。

DDF 的状态是指令 DISPLAY ON/OFF 和 RST 信号控制的。

（5）XY 地址计数器

XY 地址计数器是一个 9 位计数器。高 3 位是 X 地址计数器，低 6 位为 Y 地址计数器，XY 地址计数器实际上是作为 DDRAM 的地址指针，X 地址计数器为 DDRAM 的页指针，Y 地址计数器为 DDRAM 的 Y 地址指针。

X 地址计数器是没有计数功能的，只能用指令设置。

Y 地址计数器具有循环计数功能，各显示数据写入后，Y 地址自动加 1，Y 地址指针从 0 到 63。

（6）显示数据 RAM（DDRAM）

DDRAM 是存储图形显示数据的。数据为 1 表示显示选择，数据为 0 表示显示非选择。

（7）Z 地址计数器

Z 地址计数器是一个 6 位计数器，此计数器具备循环计数功能，它是用于显示行扫描同

步。当一行扫描完成，此地址计数器自动加 1，指向下一行扫描数据，RST 复位后 Z 地址计数器为 0。

Z 地址计数器可以用指令 DISPLAY START LINE 预置。因此，显示屏幕的起始行就由此指令控制，即 DDRAM 的数据从哪一行开始显示在屏幕的第 1 行。此模块的 DDRAM 共 64 行，屏幕可以循环滚动显示 64 行。

3. 12864LCD 的指令系统及时序

该类液晶显示模块（即 KS0108B 及其兼容控制驱动器）的指令系统比较简单，总共只有 7 种。其指令表见表 11-20。

表 11-20　12864LCD 指令表

指令名称	控制信号		控制代码							
	R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
显示开关	0	0	0	0	1	1	1	1	1	1/0
显示起始行设置	0	0	1	1	X	X	X	X	X	X
页设置	0	0	1	0	1	1	1	X	X	X
列地址设置	0	0	0	1	X	X	X	X	X	X
读状态	1	0	BUSY	0	ON/OFF	RST	0	0	0	0
写数据	0	1	写数据							
读数据	1	1	读数据							

各功能指令分别介绍如下。

（1）显示开/关指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	1	1	1	1	1/0

当 DB0 = 1 时，LCD 显示 RAM 中的内容；DB0 = 0 时，关闭显示。

（2）显示起始行（ROW）设置指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	1	显示起始行(0 ~ 63)					

该指令设置了对应液晶屏最上一行的显示 RAM 的行号，有规律地改变显示起始行，可以使 LCD 实现显示滚屏的效果。

（3）页（PAGE）设置指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	0	1	1	1	页号(0 ~ 7)		

显示 RAM 共 64 行，分 8 页，每页 8 行。

（4）列地址（Y Address）设置指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	显示列地址(0 ~ 63)					

设置了页地址和列地址，就唯一确定了显示 RAM 中的一个单元，这样 MPU 就可以用读、写指令读出该单元中的内容或向该单元写进一个字节数据。

（5）读状态指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	BUSY	0	ON/OFF	REST	0	0	0	0

该指令用来查询液晶显示模块内部控制器的状态，各参量含义如下：

BUSY：1-内部在工作，0-正常状态。

ON/OFF：1-显示关闭，0-显示打开。

RESET：1-复位状态，0-正常状态。

在 BUSY 和 RESET 状态时，除读状态指令外，其他指令均不对液晶显示模块产生作用。在对液晶显示模块操作之前要查询 BUSY 状态，以确定是否可以对液晶显示模块进行操作。

（6）写数据指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	写　数　据							

（7）读数据指令

R/W	RS	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	读　显　示　数　据							

读、写数据指令每执行完一次读、写操作，列地址就自动增 1。必须注意的是，进行读操作之前，必须有一次空读操作，紧接着再读才会读出所要读的单元中的数据。

LCD 基本读/写操作时序如图 11-66 所示。

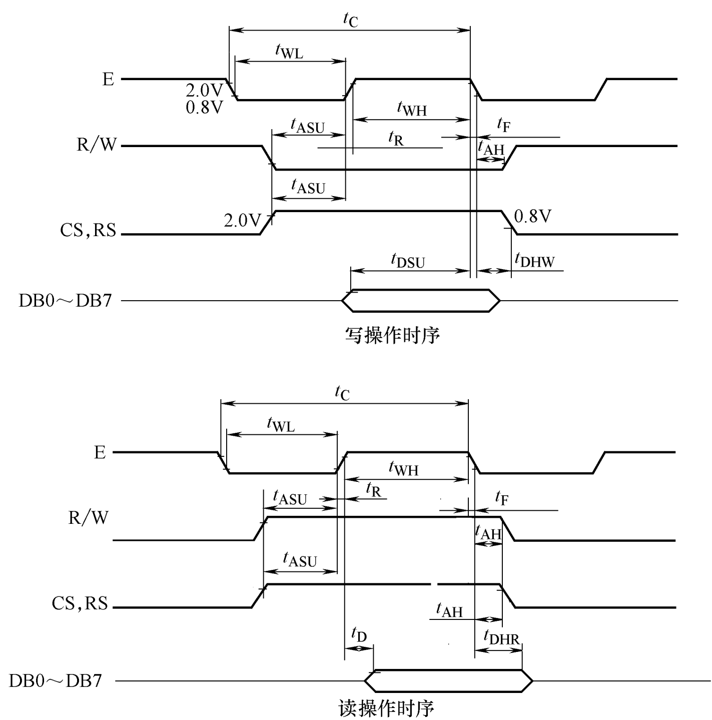


图 11-66　读/写操作时序

11.9.3　12864 点阵型 LCD 软硬件设计实例

通过以上学习，现在就来实际进行应用 12864LCD 的软硬件设计。本实例将在 LCD 上显示如图 11-67 所示内容。

				欢		迎		使		用					
		单		片		机		开		发		板			
当		前		状		态		:	运		行		中		
	w	w	w	.	h	i	f	i	c	a	t	.	c	o	m

图 11-67　模拟显示效果图

1. 硬件原理图（见图 11-68）

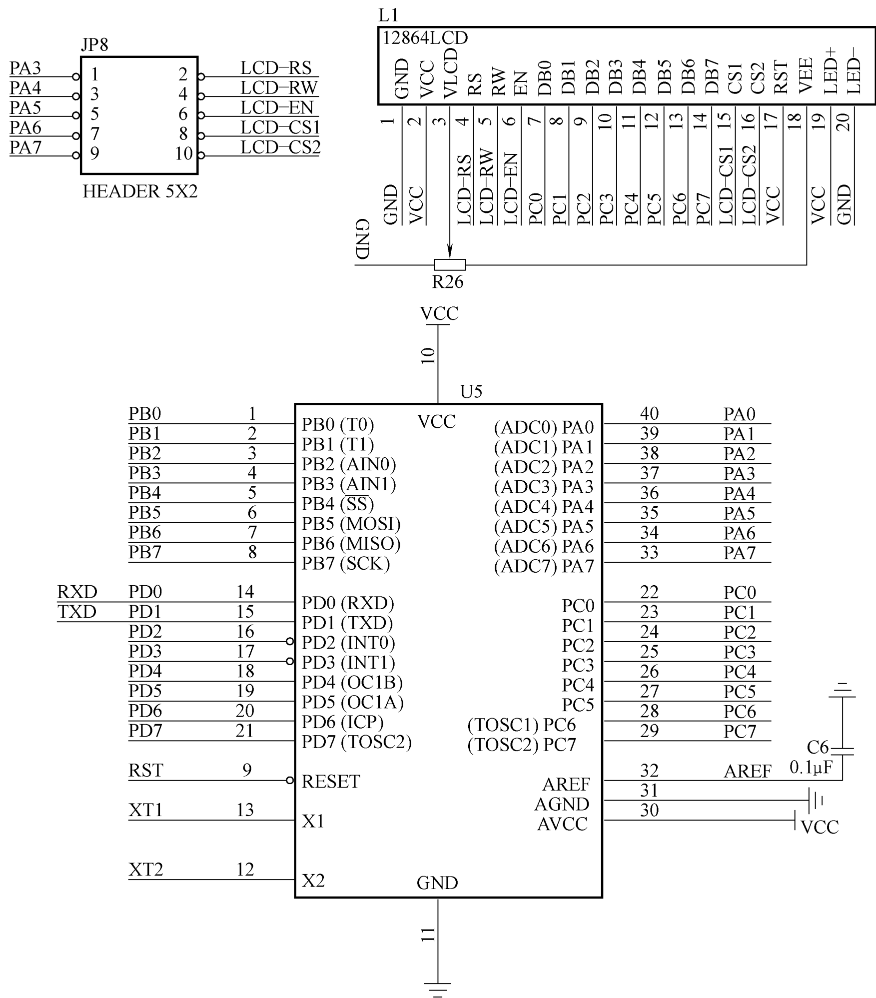


图 11-68　硬件原理图

2. 软件设计

在编写软件代码之前必须要先掌握汉字取模的方法。要得到图 11-67 中的文字，可以借助取模软件来完成。目前点阵 LCD 的取模软件有很多，以本开发板配套的取模软件为例来介绍一下汉字的取模方法。

打开取模软件出现如图 11-69 所示的显示界面。

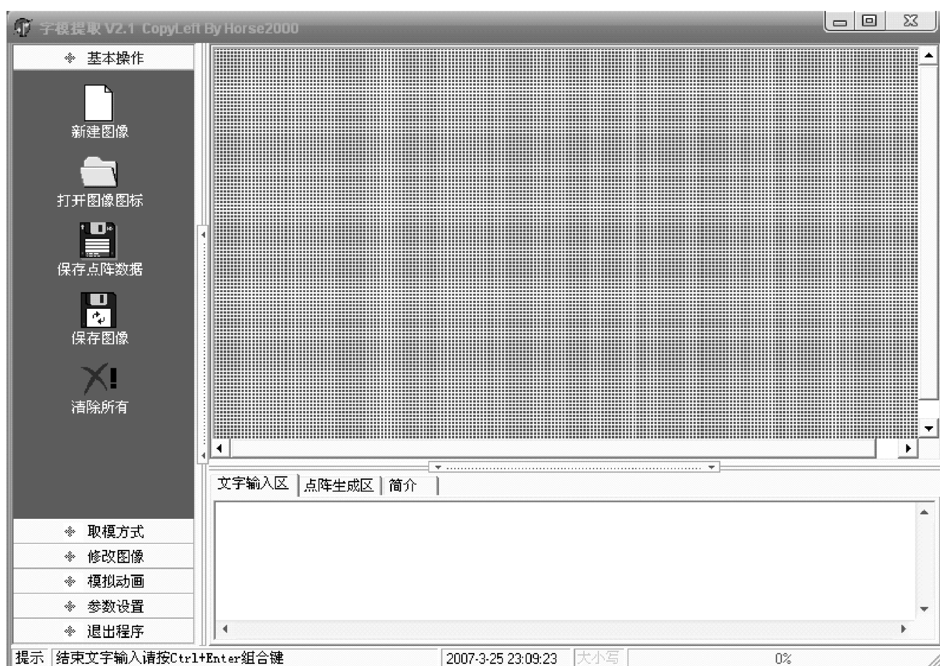


图 11-69　显示界面

在文字输入区中输入文字，以输入一个“欢”字为例，了解其取模过程。在文字输入区中输入“欢”，按 CTRL + ENTER 组合键后就看到“欢”字已经在模拟显示区显示出来了，如图 11-70 所示。

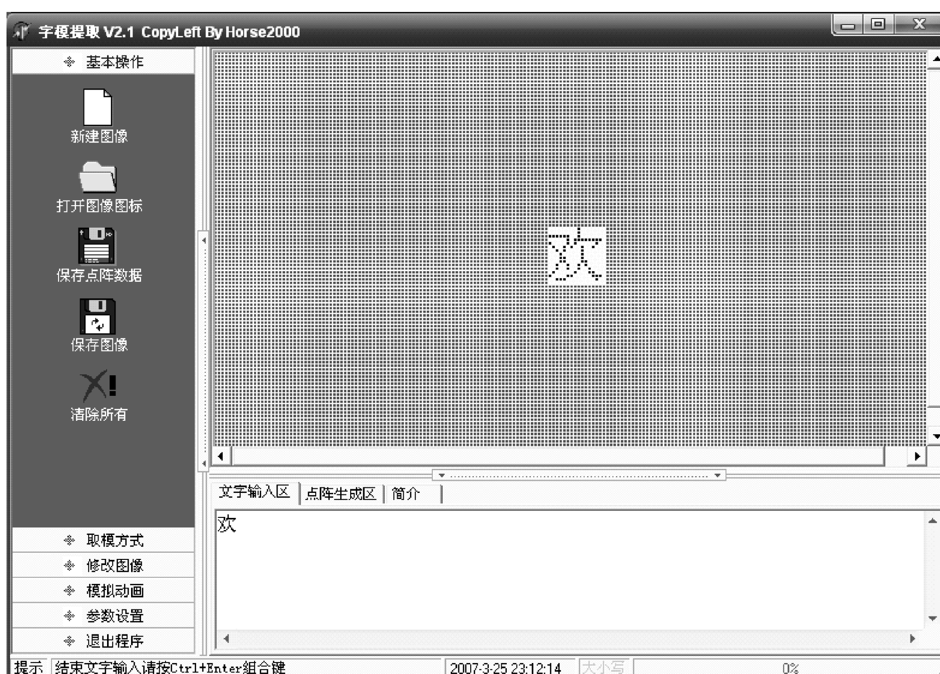


图 11-70　在文字输入区中输入文字

在“取模方式”中选择“C51 格式”就可以在“点阵生成区”得到需要的汉字“欢”

的显示代码，如图 11-71 所示。

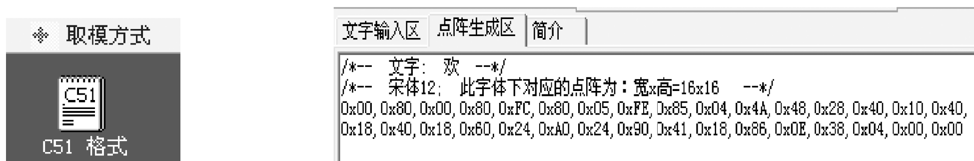


图 11-71　生成汉字的显示代码

经过以上步骤后一个汉字就取模成功了，在程序中只要调用这段代码就可显示出汉字“欢”了，其他汉字也用同样的方法。取完要显示的全部汉字代码后就可以编程了，最终的软件代码如下：

```
/*****************************************************************************/
/*128×64LCD 测试程序                                                          */
/*目标器件:ATmega16                                                           */
/*晶振:RC 1MHz                                                                */
/*编译环境:ICCAVR 6.31A                                                       */
/*****************************************************************************/

/********************************** 包含头文件 *********************************/
#include <iom16v.h>
#include <macros.h>

/********************************** 命令字定义 *********************************/
#define Disp_On      0x3f
#define Disp_Off     0x3e
#define Col_Add      0x40
#define Page_Add     0xb8
#define Start_Line   0xc0

/********************************** 端口定义 ***********************************/
#define    Mcson          PORTA |= BIT(6);
#define    Mcsoff         PORTA &= ~BIT(6);

#define    Scson          PORTA |= BIT(7);
#define    Scsoff         PORTA &= ~BIT(7);

#define    Enon           PORTA |= BIT(5);
#define    Enoff          PORTA &= ~BIT(5);

#define    Rson           PORTA |= BIT(3);
#define    Rsoff          PORTA &= ~BIT(3);
```

```
#define     Rwon          PORTA |=BIT(4);
#define     Rwoff         PORTA &= ~BIT(4);

/*********************************** 字模表 ************************************/
/*************************** www.hificat.com ********************************/
const unsigned char h[] = {
/*--  文字:  h  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x08,0xF8,0x00,0x80,0x80,0x80,0x00,0x00,0x20,0x3F,0x21,0x00,0x00,0x20,0x3F,0x20,
};
const unsigned char w[] = {
/*--  文字:  w  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x80,0x80,0x00,0x80,0x00,0x80,0x80,0x80,0x0F,0x30,0x0C,0x03,0x0C,0x30,0x0F,0x00,
};
const unsigned char i[] = {
/*--  文字:  i  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x80,0x98,0x98,0x00,0x00,0x00,0x00,0x00,0x20,0x20,0x3F,0x20,0x20,0x00,0x00,
};
const unsigned char f[] = {
/*--  文字:  f  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x80,0x80,0xF0,0x88,0x88,0x88,0x18,0x00,0x20,0x20,0x3F,0x20,0x20,0x00,0x00,
};
const unsigned char c[] = {
/*--  文字:  c  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x00,0x00,0x80,0x80,0x80,0x00,0x00,0x00,0x0E,0x11,0x20,0x20,0x20,0x11,0x00,
};
const unsigned char a[] = {
/*--  文字:  a  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x00,0x80,0x80,0x80,0x80,0x00,0x00,0x00,0x19,0x24,0x22,0x22,0x22,0x3F,0x20,
};
const unsigned char t[] = {
/*--  文字:  t  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x80,0x80,0xE0,0x80,0x80,0x00,0x00,0x00,0x00,0x00,0x1F,0x20,0x20,0x00,0x00,
```

```
};
const unsigned char o[ ] = {
/*--  文字:  o  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x00,0x80,0x80,0x80,0x80,0x00,0x00,0x00,0x1F,0x20,0x20,0x20,0x20,0x1F,0x00,
};
const unsigned char m[ ] = {
/*--  文字:  m  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x80,0x80,0x80,0x80,0x80,0x80,0x80,0x00,0x20,0x3F,0x20,0x00,0x3F,0x20,0x00,0x3F,
};
const unsigned char dian[ ] = {
/*--  文字:  .  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x30,0x30,0x00,0x00,0x00,0x00,0x00,
};
/************************************** 欢迎使用 ********************************/
const unsigned char huan[ ] = {
/*--  文字:  欢  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x14,0x24,0x44,0x84,0x64,0x1C,0x20,0x18,0x0F,0xE8,0x08,0x08,0x28,0x18,0x08,0x00,
0x20,0x10,0x4C,0x43,0x43,0x2C,0x20,0x10,0x0C,0x03,0x06,0x18,0x30,0x60,0x20,0x00,
};
const unsigned char yun2[ ] = {
/*--  文字:  迎  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x40,0x41,0xCE,0x04,0x00,0xFC,0x04,0x02,0x02,0xFC,0x04,0x04,0x04,0xFC,0x00,0x00,
0x40,0x20,0x1F,0x20,0x40,0x47,0x42,0x41,0x40,0x5F,0x40,0x42,0x44,0x43,0x40,0x00,
};
const unsigned char shi[ ] = {
/*--  文字:  使  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x40,0x20,0xF0,0x1C,0x07,0xF2,0x94,0x94,0x94,0xFF,0x94,0x94,0x94,0xF4,0x04,0x00,
0x00,0x00,0x7F,0x00,0x40,0x41,0x22,0x14,0x0C,0x13,0x10,0x30,0x20,0x61,0x20,0x00,
};
const unsigned char yong[ ] = {
/*--  文字:  用  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x00,0x00,0x00,0xFE,0x22,0x22,0x22,0x22,0xFE,0x22,0x22,0x22,0x22,0xFE,0x00,0x00,
0x80,0x40,0x30,0x0F,0x02,0x02,0x02,0x02,0xFF,0x02,0x02,0x42,0x82,0x7F,0x00,0x00,
```

```
};

/*********************************** 单片机开发板 *****************************/
const unsigned char dan[] = {
/*--  文字:  单  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x00,0x00,0xF8,0x28,0x29,0x2E,0x2A,0xF8,0x28,0x2C,0x2B,0x2A,0xF8,0x00,0x00,0x00,
0x08,0x08,0x0B,0x09,0x09,0x09,0x09,0xFF,0x09,0x09,0x09,0x09,0x0B,0x08,0x08,0x00,
};
const unsigned char pian[] = {
///*--  文字:  片  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x00,0x00,0x00,0xFE,0x10,0x10,0x10,0x10,0x10,0x1F,0x10,0x10,0x10,0x18,0x10,0x00,
0x80,0x40,0x30,0x0F,0x01,0x01,0x01,0x01,0x01,0x01,0x01,0xFF,0x00,0x00,0x00,0x00,
};
const unsigned char ji[] = {
/*--  文字:  机  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x08,0x08,0xC8,0xFF,0x48,0x88,0x08,0x00,0xFE,0x02,0x02,0x02,0xFE,0x00,0x00,0x00,
0x04,0x03,0x00,0xFF,0x00,0x41,0x30,0x0C,0x03,0x00,0x00,0x00,0x3F,0x40,0x78,0x00,
};
const unsigned char kai[] = {
/*--  文字:  开  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x40,0x42,0x42,0x42,0x42,0xFE,0x42,0x42,0x42,0x42,0xFE,0x42,0x42,0x42,0x42,0x00,
0x00,0x40,0x20,0x10,0x0C,0x03,0x00,0x00,0x00,0x00,0x7F,0x00,0x00,0x00,0x00,0x00,
};
const unsigned char fa[] = {
/*--  文字:  发  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x00,0x10,0x3E,0x10,0x10,0xF0,0x9F,0x90,0x90,0x92,0x94,0x1C,0x10,0x10,0x10,0x00,
0x40,0x20,0x10,0x88,0x87,0x41,0x46,0x28,0x10,0x28,0x27,0x40,0xC0,0x40,0x00,0x00,
};
const unsigned char ban[] = {
/*--  文字:  板  --*/
/*--  宋体12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x10,0x10,0xD0,0xFF,0x50,0x90,0x00,0xFE,0x62,0xA2,0x22,0x21,0xA1,0x61,0x00,0x00,
0x04,0x03,0x00,0x7F,0x00,0x11,0x0E,0x41,0x20,0x11,0x0A,0x0E,0x31,0x60,0x20,0x00,
};
```

```
const unsigned char dang[ ] = {
/*--  文字：  当  --*/
/*--  宋体 12；  此字体下对应的点阵为：宽×高 = 16×16  --*/
0x00,0x00,0x40,0x42,0x5C,0x48,0x40,0x40,0x7F,0x40,0x50,0x4E,0x44,0xC0,0x00,0x00,
0x00,0x00,0x20,0x22,0x22,0x22,0x22,0x22,0x22,0x22,0x22,0x22,0x22,0x7F,0x00,0x00,
};

const unsigned char qian[ ] = {
/*--  文字：  前  --*/
/*--  宋体 12；  此字体下对应的点阵为：宽×高 = 16×16  --*/
0x08, 0x08, 0xE8, 0xA8, 0xA9, 0xAE, 0xEA, 0x08, 0x08, 0xC8, 0x0C, 0x0B, 0xEA, 0x08,
0x08,0x00,
0x00,0x00,0x7F,0x04,0x24,0x44,0x3F,0x00,0x00,0x1F,0x40,0x80,0x7F,0x00,0x00,0x00,
};

const unsigned char zhuang[ ] = {
/*--  文字：  状  --*/
/*--  宋体 12；  此字体下对应的点阵为：宽×高 = 16×16  --*/
0x08,0x30,0x00,0xFF,0x20,0x20,0x20,0x20,0xFF,0x20,0xE1,0x26,0x2C,0x20,0x20,0x00,
0x04,0x02,0x01,0xFF,0x40,0x20,0x18,0x07,0x00,0x00,0x03,0x0C,0x30,0x60,0x20,0x00,
};

const unsigned char tai[ ] = {
/*--  文字：  态  --*/
/*--  宋体 12；  此字体下对应的点阵为：宽×高 = 16×16  --*/
0x00,0x04,0x04,0x04,0x84,0x44,0x34,0x4F,0x94,0x24,0x44,0x84,0x84,0x04,0x00,0x00,
0x00,0x60,0x39,0x01,0x00,0x3C,0x40,0x42,0x4C,0x40,0x40,0x70,0x04,0x09,0x31,0x00,
};

const unsigned char yun[ ] = {
/*--  文字：  运  --*/
/*--  宋体 12；  此字体下对应的点阵为：宽×高 = 16×16  --*/
0x40,0x41,0xCE,0x04,0x00,0x20,0x22,0xA2,0x62,0x22,0xA2,0x22,0x22,0x22,0x20,0x00,
0x40,0x20,0x1F,0x20,0x28,0x4C,0x4A,0x49,0x48,0x4C,0x44,0x45,0x5E,0x4C,0x40,0x00,
};

const unsigned char xing[ ] = {
/*--  文字：  行  --*/
/*--  宋体 12；  此字体下对应的点阵为：宽×高 = 16×16  --*/
0x10,0x08,0x84,0xC6,0x73,0x22,0x40,0x44,0x44,0x44,0xC4,0x44,0x44,0x44,0x40,0x00,
```

```
0x02,0x01,0x00,0xFF,0x00,0x00,0x00,0x00,0x40,0x80,0x7F,0x00,0x00,0x00,0x00,0x00,
};

const unsigned char zhong[] = {
/*--  文字:  中  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=16×16  --*/
0x00,0x00,0xFC,0x08,0x08,0x08,0x08,0xFF,0x08,0x08,0x08,0x08,0xFC,0x08,0x00,0x00,
0x00,0x00,0x07,0x02,0x02,0x02,0x02,0xFF,0x02,0x02,0x02,0x02,0x07,0x00,0x00,0x00,
};

const unsigned char maohao[] = {
/*--  文字:  :  --*/
/*--  宋体 12;  此字体下对应的点阵为:宽×高=8×16  --*/
0x00,0x00,0x00,0xC0,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x30,0x30,0x00,0x00,0x00,
};

/*****************************************************************************
函数功能:延时程序
入口参数:t
出口参数:
*****************************************************************************/
void delay(unsigned int t)
{
    unsigned int i,j;
    for(i=0;i<t;i++)
    for(j=0;j<10;j++);
}

/*****************************************************************************
函数功能:写命令到 LCD 程序
入口参数:cmdcode
出口参数:
*****************************************************************************/
void write_com(unsigned char cmdcode)
{
    Rsoff;
    Rwoff;
    PORTC=cmdcode;
    delay(2);
    Enon;
```

```
    delay(2);
    Enoff;
}

/ ****************************************************************************
函数功能:写数据到 LCD 程序
入口参数:Dispdata
出口参数:
 **************************************************************************** /
void write_data(unsigned char Dispdata)
{
    Rson;
    Rwoff;
    PORTC = Dispdata;
    delay(2);
    Enon;
    delay(2);
    Enoff;
}

/ ****************************************************************************
函数功能:清除 LCD 内存程序
入口参数:pag,col,hzk
出口参数:
 **************************************************************************** /
void Clr_Scr()
{
    unsigned char j,k;
    Mcson;Scson;
    write_com(Page_Add +0);
    write_com(Col_Add +0);
    for(k =0;k <8;k + +)
    {
        write_com(Page_Add + k);
        for(j =0;j <64;j + +)write_data(0x00);
    }
    Mcsoff;Scsoff;
}

/ ****************************************************************************
```

```
函数功能:指定位置显示数字 16 ×16 程序
入口参数:pag,col,hzk
出口参数:
*******************************************************************************/
void hz_disp16(unsigned char pag,unsigned char col,const unsigned char *hzk)
{
    unsigned char j = 0,i = 0;
    for(j = 0;j < 2;j++)
    {
        write_com(Page_Add + pag + j);
        write_com(Col_Add + col);
        for(i = 0;i < 16;i++)
        write_data(hzk[16 * j + i]);
    }
}

/*******************************************************************************
函数功能:指定位置显示数字 8 ×16 程序
入口参数:pag,col,hzk
出口参数:
*******************************************************************************/
void hz_disp8(unsigned char pag,unsigned char col,const unsigned char *hzk)
{
    unsigned char j = 0,i = 0;
    for(j = 0;j < 2;j++)
    {
        write_com(Page_Add + pag + j);
        write_com(Col_Add + col);
        for(i = 0;i < 8;i++)
        write_data(hzk[8 * j + i]);
    }
}
/*******************************************************************************
函数功能:LCD 初始化程序
入口参数:
出口参数:
*******************************************************************************/
void init_lcd()
{
    delay(100);
```

```
    Mcson;
    Scson;
    delay(100);
    write_com(Disp_Off);
    write_com(Page_Add +0);
    write_com(Start_Line +0);
    write_com(Col_Add +0);
    write_com(Disp_On);
}

/ ******************************************************************************
函数功能:主程序
入口参数:
出口参数:
 ****************************************************************************** /
void main(void)
{
    DDRA =0xff;
    DDRC =0xff;
    init_lcd();
    Clr_Scr();
    Mcson;Scsoff;           //左、右都显示
    while(1)
    {
        Mcson;Scsoff;       //左显示
        delay(2);
        //欢迎
        hz_disp16(0,32,huan);
        hz_disp16(0,48,yun2);
        //单片机
        hz_disp16(2,16,dan);
        hz_disp16(2,32,pian);
        hz_disp16(2,48,ji);
        //当前状态
        hz_disp16(4,0,dang);
        hz_disp16(4,16,qian);
        hz_disp16(4,32,zhuang);
        hz_disp16(4,48,tai);
        //网址:www. hifi
```

```
        hz_disp8(6,0,w);
        hz_disp8(6,8,w);
        hz_disp8(6,16,w);
        hz_disp8(6,24,dian);
        hz_disp8(6,32,h);
        hz_disp8(6,40,i);
        hz_disp8(6,48,f);
        hz_disp8(6,56,i);

        Mcsoff;Scson;            //右显示
        //使用
        hz_disp16(0,0,shi);
        hz_disp16(0,16,yong);
        //开发板
        hz_disp16(2,0,kai);
        hz_disp16(2,16,fa);
        hz_disp16(2,32,ban);
        //:运行中
        hz_disp8(4,0,maohao);
        hz_disp16(4,8,yun);
        hz_disp16(4,24,xing);
        hz_disp16(4,40,zhong);
        //网址:cat.com
        hz_disp8(6,0,c);
        hz_disp8(6,8,a);
        hz_disp8(6,16,t);
        hz_disp8(6,24,dian);
        hz_disp8(6,32,c);
        hz_disp8(6,40,o);
        hz_disp8(6,48,m);
        delay(2);
    }
}
```

11.10　红外遥控软件解码应用实例

11.10.1　红外遥控概述

红外遥控是一种无线、非接触式的远程控制技术。因其具有体积小、功耗低、抗干扰能

力强、成本低、易于实现等优点，已被广泛应用于各类家电和电子产品中，如电视机、VCD/DVD 机、空调、手机、笔记本电脑等。图 11-72 所示为一些常见红外遥控器实物图。

目前生产红外遥控器的厂商有很多，不同厂商生产的产品也需要采用不同的代码或者编码，所以目前实际使用的编码方法非常多，常见的有 30 种以上，例如最常见的有 3010、6121、7461、9012 等。一般现在的一些单功能编码芯片还是用数字电路实现的，但是多功能的则大多用 4 位或 8 位单片机实现。通常不同的家用电器的红外遥控器采用不同的编码芯片，以此来避免不同电器的遥控干扰问题。



图 11-72　一些常见红外遥控器实物图

1．红外线

红外线又称红外光波。在电磁波谱中，光波的波长范围为 0.01 ~1000μm。根据波长的不同可分为可见光和不可见光，波长为 0.38 ~0.76μm 的光波称为可见光，依次为红、橙、黄、绿、青、蓝、紫 7 种颜色。光波为 0.01 ~0.38μm 的光波为紫外光（线），波长为 0.76 ~1000μm 的光波为红外光（线）。光的波长分布如图 11-73 所示。

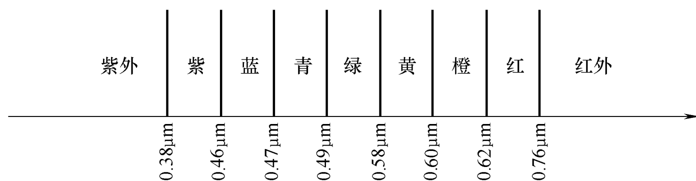


图 11-73　光波长分布图

红外光按波长范围分为近红外、中红外、远红外、极红外 4 类。红外线遥控是利用近红外光传送遥控指令的，波长为 0.76 ~1.5μm。用近红外光作为遥控光源，是因为目前红外发射器件（红外发光管）与红外接收器件（光敏二极管、光敏晶体管及光电池）的发光与受光峰值波长一般为 0.8 ~0.94μm，在近红外光波段内，两者的光谱正好重合，能够很好地匹配，可以获得较高的传输效率及较高的可靠性。

2．红外遥控原理

红外遥控系统一般由红外发射装置和红外接收设备两大部分组成（见图 11-74）。红外发射装置由键盘电路、红外编码芯片、电源和红外发射电路组成。红外接收设备可由红外接收电路、红外解码芯片、电源和应用电路组成。

通常为了使信号能更好地被传输发送端，将基带二进制信号调制为脉冲串信号，通过红外发射管发射。常用的有，通过脉冲宽度来实现信号调制的脉宽调制（PWM）和通过脉冲串之间的时间间隔来实现信号调制的脉时调制（PPM）两种方法。

（1）红外信号的产生

前面已经提到红外线的波长范围为 0.76 ~1.5μm，而且用近红外光作为遥控光源，不过在

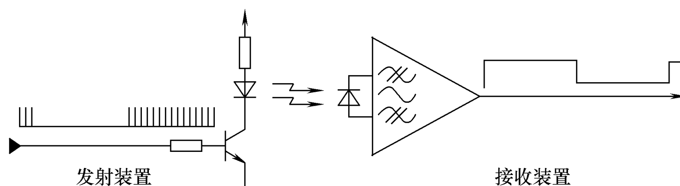


图 11-74　红外遥控系统组成

红外遥控中，通常使用的波长是 0.94μm，一般用来产生这种信号的器件是红外发光二极管。红外发光二极管使用的材料不同于普通发光二极管，因而在红外发光二极管两端施加一定电压时，它发出的是红外线而不是可见光。图 11-75 所示为两种红外发光二极管的实物图。

图 11-75　两种常见的红外发光二极管实物图

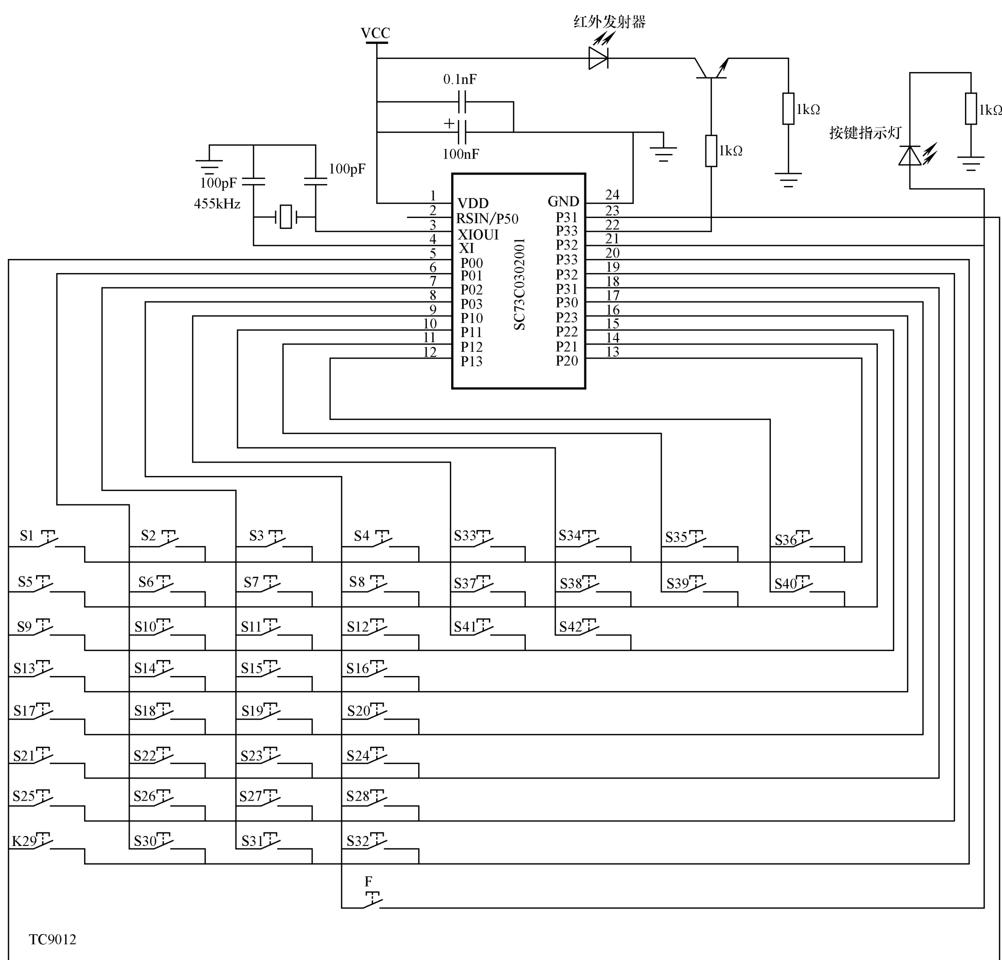


图 11-76　某成品遥控器电路原理图

红外发光二极管一般有黑色、深蓝、透明 3 种颜色。判断红外发光二极管好坏的办法与判断普通二极管一样：用万用表电阻挡量一下红外发光二极管的正、反向电阻即可。红外发光二极管的驱动电路很简单，图 11-76 所示是一个成品遥控器的原理图，从图中可以看出，红外发光二极管的驱动电路主要就是一个 NPN 型管，再辅以简单的外围电路即可。

（2）红外信号的接收

用来接收红外信号的器件也是一种二极管，与可见光光敏二极管相比，其特点就是对红外信号敏感，如图 11-77a 所示 3 个图片就是 3 种常见的红外接收二极管的外形，它们最重要的特性就是，当它们处在反向偏置状态时，反向电流随着受到的红外信号强度的增强而上升。根据这一特性，就可以得到如图 11-77b 所示的一个简单的信号接收电路。

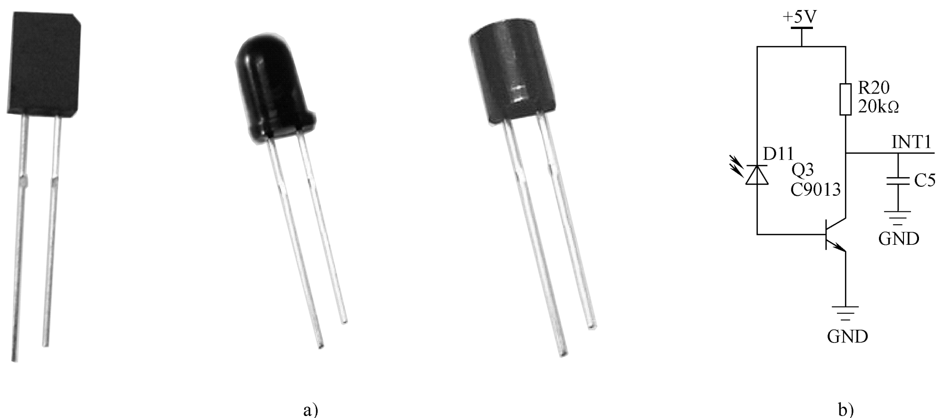


图 11-77　3 种常见的红外接收二极管实物图和简易接收电路

在图 11-77b 的接收电路里，红外接收二极管反向偏置直接串在晶体管基极和电源正端之间，这样在晶体管的集电极就可以产生一个反相变化的信号，由于红外发光二极管的发射功率一般都较小（100mW 左右），而且接收管的灵敏度也很有限，所以如果要在晶体管集电极产生满足逻辑电平条件的信号，那么这个接收电路的实际工作距离只有 10 ~ 20cm 左右（信号由普通红外遥控器产生），这显然不能满足实际使用要求。为了增加接收距离，一般要增加高增益放大电路，然而因为增益的增加会使得接收电路更容易受到干扰，所以也必须有相应的防干扰电路。实际应用中，带高增益放大电路的接收电路已经很成熟，比如以前电视机的红外遥控接收电路里常用的 CX20106A 红外接收专用集成放大电路，如图 11-78 所示。

CX20106A 在一个电路里集成了前置放大、限幅放大、带通滤波、波形整形等电路，实际使用时可以调整放大倍数、带通滤波的中心频率等参数。因为抗干扰的需要，整个电路需要封装在一个金属屏蔽盒里，因而结构比较复杂，体积也不小。

最近几年不论是业余制作还是正式产品，大多都采用一体化红外接收头。一体化接收头是把上面所有的电路都集成在一个封装里，体积更小，可靠性更高。一体化红外接收头的封装有两种：一种采用铁皮屏蔽；一种是塑料封装，均只有 3 只引脚，即电源正（VDD）、电源负（GND）和数据输出（VO 或 OUT）。每种封装又分大小，小的如图 11-79a 所示，大的如图 11-79b 所示。红外接收头的引脚排列因型号不同而不尽相同，可参考厂商的使用说明。一体化红外接收头不需要复杂的调试和外壳屏蔽，使用起来如同一只晶体管，非常方便，但

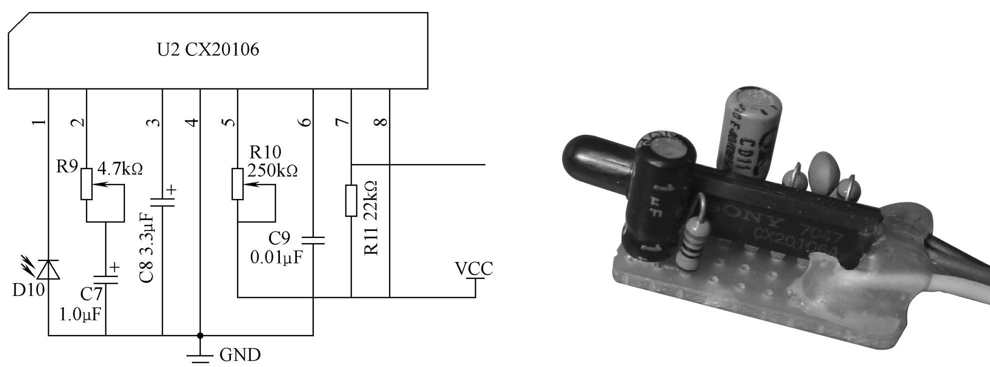


图 11-78　CX20106A 的原理图和简单的实物图

在使用时注意成品红外接收头的载波频率。红外遥控最常用的载波频率为 38kHz，这是由发射端使用的 455kHz 晶振决定的，也有一些遥控系统采用 36kHz、40kHz、56kHz 等。

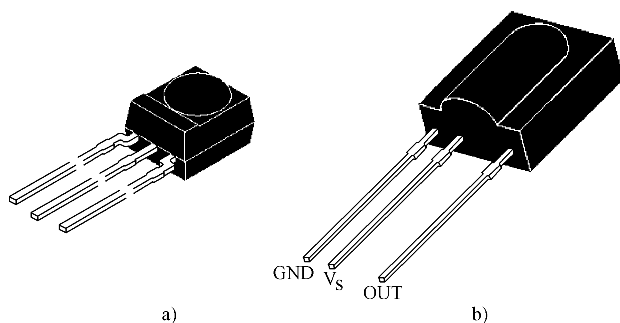


图 11-79　两种常见的红外接收二极管实物图

一体化红外接收头由于封装是固定的，所以能够接收的频率是固定的，增益的调整也不再通过外部电路实现，而是在内部使用了增益自动控制（AGC）电路。图 11-80 所示为 TSOP173x 系列一体化红外接收头的原理框图。

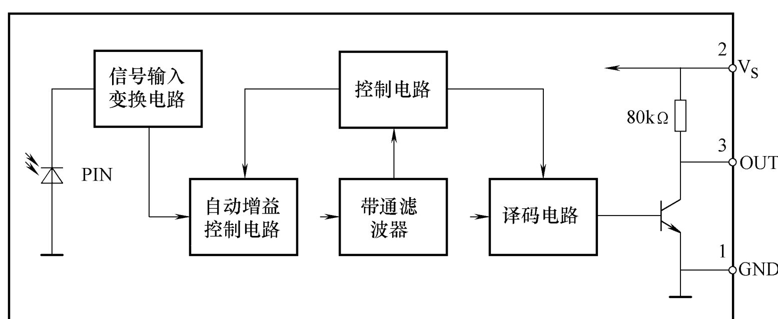


图 11-80　TSOP173x 系列一体化红外接收头的原理框图

（3）红外信号的编码和解码

在同一个遥控电路中通常要实现不同的遥控功能或区分不同的机器类型，这样就要求信号要按一定的编码传送。编码会由编码芯片或电路完成；对应于编码芯片通常会有相配对的解码芯片或包含解码模块的应用芯片。在实际的产品设计中，为了能够使用单片机或数字电

路去制定解码方案，就需要了解所使用的编码芯片的编码方式。

下面，通过分析两种典型的编码方式来说明红外信号的编码过程。

早期的红外遥控中，发射的红外信号是不经过调制的，比如一种现在仍在使用的 IRT1250 芯片的编码，编码方式如图 11-81 所示。这种编码用前后两个脉冲之间的时间长度来编码“0”和“1”，而在 10μs 的信号发射期间，红外发光二极管是一直处于导通状态的。使用发现，因为红外发光二极管长时间大电流工作，导致使用寿命不长，所以这种编码方式一般要在所加电压和导通时间之间找合适的平衡点。无载波调制的编码一般都将红外发光二极管的导通时间控制得比较小，由此带来的缺点就是遥控距离近，而且容易受到干扰，使用起来效果不好，所以后来采用了调制和解调的技术。

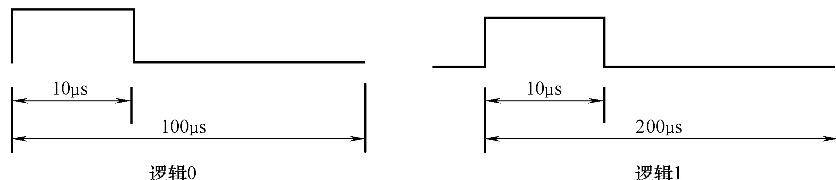


图 11-81　IRT1250 的编码

红外遥控发射器是一种脉冲编码调制器，它在发射遥控指令时，把二进制数调制成一系列的脉冲串信号后发射出去，常用的调制方法有脉冲宽度调制（PWM）和脉冲相位调制（PPM）两种，比如前面提到的 IRT1250 的编码方式就是一个很典型的脉冲宽度调制的例子，下面分别以 μPD6121（PWM）和 SAA3010（PPM）的编码为例来介绍这两种编码的方法。

图 11-82 所示的是一组按照 μPD6121 编码发射出去的一帧数据的波形。最左边部分叫做引导码，后面的 0 和 1 编码在红外发光二极管工作期间并不是一直导通的，而是以 38kHz 的频率间歇地导通和截止，这就使得加在红外发光二极管上的瞬时电压可以增大，这样遥控距离就提高了很多，而且编码中 0 和 1 的高电平时间也可以变大，而不用担心红外发光二极管会连续工作带来的寿命问题，还有非常重要的一点，就是在采用了调制和解调之后，遥控器的抗干扰能力也有了大幅度提高。

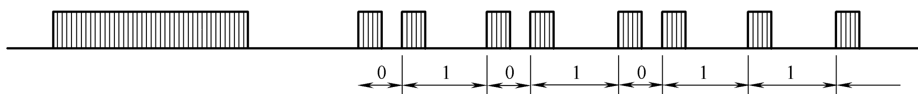


图 11-82　μPD6121 编码发射某数据时前面部分信号示意图

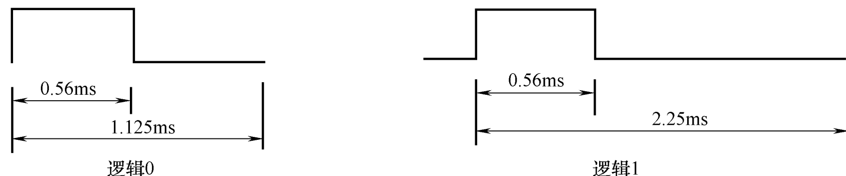


图 11-83　μPD6121 编码的逻辑 0 和 1 的定义

图 11-83 所示的是 μPD6121 编码的逻辑 0 和逻辑 1 的定义。我们可以看到，μPD6121 的编码方式，把低电平宽度为 1 个周期的信号定义为逻辑 0，低电平持续 3 个周期的定义为逻辑 1，这样通过对低电平宽度的区分就可以识别出 0 和 1，然后把遥控信号组成一个串行序列。

一帧完整的 μPD6121 的发射码有引导码、用户编码和键数据码 3 部分组成。引导码由一个 9ms 高电平脉冲及 4.5ms 的低电平脉冲组成；8 位用户编码，被连续发送两次；8 位的键数据码也被连续发送两次，第 1 次发送的是键数据码的原码，第 2 次发送的是键数据码的反码，这样就可以在接收端实现对数据的校验，具体发射码构成如图 11-84 所示。

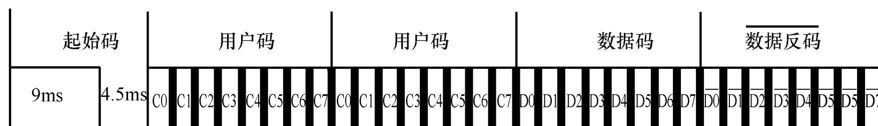


图 11-84　一帧完整的 μPD6121 的发射码构成

11.10.2　μPD6121 红外接收的软件解码应用实例

1. μPD6121 编码芯片

μPD6121 是 NEC 公司推出的一款红外遥控编码芯片。该芯片可用于低电压的场合，并且能够产生较长的防抖动时间。μPD6121 的引脚定义如图 11-85 所示。

各引脚功能如下所示：

KI0 ~ KI3：键盘输入；

REM：遥控码输出口；

VDD：电源输入；

SEL：输出码 D7 位选择，接 VDD 时 D7 为 0，接 VSS 时 D7 为 1；

KI/O0 ~ KI/O7：键盘输入/输出；

LMP：输出显示口；

CCS：用户码选择口；

OSC0、OSC1：外接晶振口。

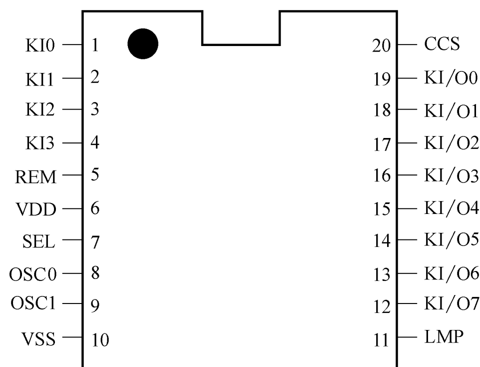


图 11-85　μPD6121 引脚定义图

2. 解码原理

由上文可知遥控码的波形图，下面介绍用单片机来解码的过程。可以用引导码的下降沿来触发单片机的定时器开始计时处理，这样就可以获得电平的时间长度。不过出于软硬件效率方面的取舍，仍然可以有不同的方法来实现，常用的连接方式有以下两种，即普通 IO 查询法和外部中断法。

注意，一般的遥控器编码在长按按键时，会连续发数据，可能是同样的数据，也有可能是特定的所谓重复帧，虽然帧间的间隔大小不等，但一般在 20 ~ 100ms 之间，而有效的 0 和 1 的编码时间却基本小于 10ms，就是说大致上 15ms 之内没有信号收到就表示当前的数据帧已经接收完毕。

11.10.3　μPD6121 解码应用设计

目前教材所配的红外遥控器为自己定制，所以按键对应码在解码时要有所区别对应。遥控器实物与对应按键码如图 11-86 所示。

1. 硬件原理设计

由于实验板上采用的外部中断法，所以下面就具体说明在用单片机外部中断的情况下实现解码过程。最终将解码内容显示在 1602LCD 上面供用户核对。硬件原理如图 11-87 所示。

00	01	02	03
04	05	06	07
08	09	0A	0B
0C	0D	0E	0F
10	11	12	13
14	15	16	17
18	19	1A	1B

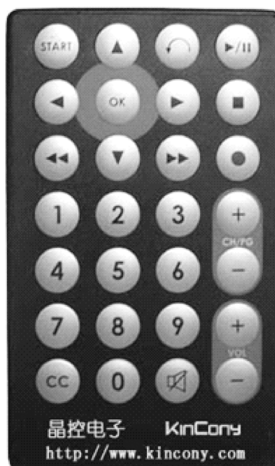


图 11-86　遥控器实物及按键对应码

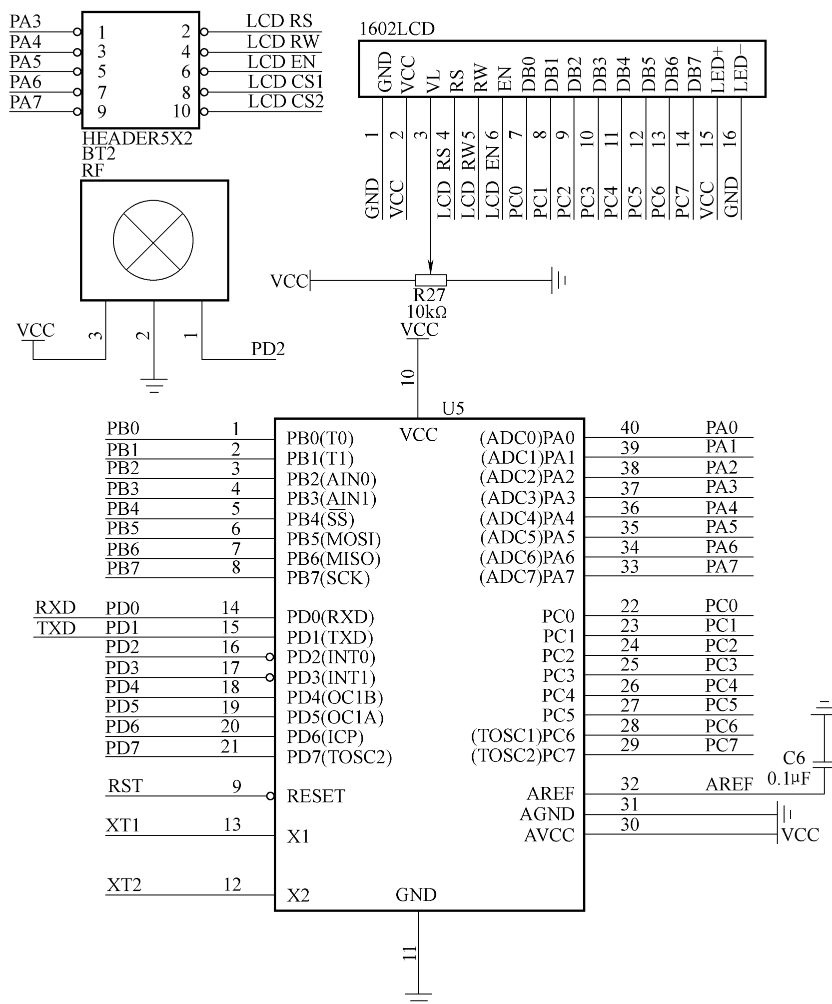


图 11-87　硬件原理图

2. 软件设计

```
/ ****************************************************************************** /
/ * 红外 HY1838 测试程序                                                         * /
/ * 目标器件:ATmega16                                                            * /
/ * 晶振:外部晶振 8MHz                                                           * /
/ * 编译环境:ICCAVR 6.31A                                                        * /
/ ****************************************************************************** /

/ ********************************** 包含头文件 ********************************* /
#include <iom16v.h>
#include <macros.h>

unsigned char i_StNum = 0;
unsigned char i_StNum1 = 0;
unsigned char FlagRF = 0;
unsigned char RF_Head = 0;
unsigned char i_RF = 0;
unsigned char code1 = 0;
unsigned char code2 = 0;
unsigned char code3 = 0;
unsigned char code4 = 0;
unsigned char code5 = 0;
unsigned char code6 = 0;

/ ********************************** 端口定义 *********************************** /
#define    Enableon     PORTA | = (1 < <5);       //EN 拉高
#define    Enableoff    PORTA & = ~(1 < <5);      //EN 拉低
#define    RSon         PORTA | = (1 < <3);       //RS 拉高
#define    RSoff        PORTA & = ~(1 < <3);      //RS 拉低
#define    RWon         PORTA | = (1 < <4);       //RW 拉高
#define    RWoff        PORTA & = ~(1 < <4);      //RW 拉低


// ================================================================
//引脚定义
// ================================================================
#define RF_IN             2  //PD2                 ID 卡信号输入

#define RF_1              PIND&(1 < <RF_IN)        //ID 为 1
#define RF_0              !(PIND&(1 < <RF_IN))     //ID 为 0
```

```
/ ******************************************************************************
函数功能:延时子程序
入口参数:k
出口参数:
****************************************************************************** /
void delay_nus(unsigned char tt)//μs 级精确延时函数,晶振 8MHz
{//简化计算 = 0.5tt + 1 (μs)
  asm("_L2: subi R16,1");
  asm("nop")
  asm("brne _L2");
  asm("nop");
  asm("ret");
}

void delay_100us(unsigned char n)
{
  unsigned char i;
  for(i=0;i<n;i++)
  {
    delay_nus(198);
  }
}

//在 8MHz 的情况
void delay_1ms(void)                    //1ms 延时函数
  {
    unsigned int i;
    for (i=0;i<1140;i++);
  }

void delay_nms(unsigned int n)          //N ms 延时函数
  {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1ms();
  }

/ ******************************************************************************
函数功能:等待 LCD 空闲程序
```

```
入口参数:
出口参数:
*************************************************************************** /
void LCD_wait(void)
{
    RSoff;//Dioff;
    RWon;
    DDRC = 0x00;                //输入
    NOP();
    Enableon;
    while(PINC&0x80);
    Enableoff;
    DDRC = 0xff;                //输出
}

/ ***************************************************************************
函数功能:LCD 写指令使能程序
入口参数:cmd
出口参数:
*************************************************************************** /
void write_command(unsigned char cmd)
{
    LCD_wait();
    RSoff;
    RWoff;
    Enableoff;
    NOP();
    Enableon;
    PORTC = cmd;
    Enableoff;
}

/ ***************************************************************************
函数功能:写数据到 LCD 程序
入口参数:dat
出口参数:
*************************************************************************** /
void write_data(unsigned char dat)
{
    LCD_wait();
```

```
    RSon;
    RWoff;
    Enableoff;
    NOP();
    Enableon;
    PORTC = dat;
    Enableoff;
}

/*****************************************************************************
函数功能:设定显示位置程序
入口参数:x,y
出口参数:
*****************************************************************************/
void Locate_XY(unsigned char x,unsigned char y)
{
    unsigned char address = (y == 0) ? (0x80 + x) : (0xc0 + x);
    write_command(address);
}

/*****************************************************************************
函数功能:指定位置显示字符串程序
入口参数:x,y, * string
出口参数:
*****************************************************************************/
void XY_String(unsigned char x,unsigned char y,unsigned char * string)
{
    unsigned int i = 0;
    Locate_XY(x,y);
    while(string[i] != '\0')
    write_data(string[i++]);
}
/*****************************************************************************
函数功能:指定位置显示数字程序
入口参数:x,y,num
出口参数:
*****************************************************************************/
void XY_Num(unsigned char x,unsigned char y,unsigned char num)
{
    Locate_XY(x,y);
```

```
    write_data(num +0x30);
}

/ ******************************************************************************
函数功能:LCD 初始化程序
入口参数:
出口参数:
 ****************************************************************************** /
void lcd_init(void)
{
    write_command(0x38);
    write_command(0x08);
    write_command(0x01);
    write_command(0x06);
    write_command(0x0c);
}



/ ******************************************************************************
函数功能:RF 接收引导码程序
入口参数:
出口参数:
 ****************************************************************************** /
void RecvRFStart(void)
{

RF1:
  i_StNum =0;
  i_StNum1 =0;
  //找结束的 1 电平
  if(RF_1)
  {
    while(RF_1);//等待 1 电平结束
  }
  else
    goto RF1;
  while(RF_0)
  {
    delay_nms(1);
```

```
    i_StNum + + ;
  }
  if( i_StNum > 8 )//9s
  {
    while( RF_1 )
    {
      delay_100us( 5 ) ;
      i_StNum1 + + ;
    }
    if( i_StNum1 > 8 )//4. 5s   引导码
    {
      RF_Head = 1 ;
    }
    else
      RF_Head = 0;
  }
  else goto RF1 ;//没找到 1 电平,错误,重新寻找
}

/ ***************************************************************************
函数功能:RF 接收引用户码和键码程序
入口参数:
出口参数:
*************************************************************************** /
void RecvRF( void)
{

      for( i_RF = 0;i_RF < 8;i_RF + + )//接受 8 位用户码
      {
        code1 = ( code1 > > 1 ) ;
        while( RF_0 ) ;//等待 0. 56ms 的低电平结束
        delay_100us( 8 ) ;//延迟 800μs 检测电平
        if( RF_1 )
        {
          code1 = ( code1 |0x80 ) ;
          while( RF_1 ) ;
        }
        else
          code1 = ( code1 |0x00 ) ;
```

```
}
for(i_RF =0;i_RF <8;i_RF + + )//接受 8 位用户反码
{
   code2 = ( code2 > >1 ) ;
   while( RF_0) ;//等待 0. 56ms 的低电平结束
   delay_100us(8) ;//延迟 800μs 检测电平
   if( RF_1)
   {
      code2 = ( code2|0x80) ;
      while( RF_1) ;
   }
   else
      code2 = ( code2|0x00) ;
}
for(i_RF =0;i_RF <8;i_RF + + )//接受 8 位键码
{
   code3 = ( code3 > >1 ) ;
   while( RF_0) ;//等待 0. 56ms 的低电平结束
   delay_100us(8) ;//延迟 800μs 检测电平
   if( RF_1)
   {
      code3 = ( code3|0x80) ;
      while( RF_1) ;
   }
   else
      code3 = ( code3|0x00) ;
}
for(i_RF =0;i_RF <8;i_RF + + )//接受 8 位键反码
{
   code4 = ( code4 > >1 ) ;
   while( RF_0) ;//等待 0. 56ms 的低电平结束
   delay_100us(8) ;//延迟 800μs 检测电平
   if( RF_1)
   {
      code4 = ( code4|0x80) ;
      while( RF_1) ;
   }
   else
      code4 = ( code4|0x00) ;
}
```

```
}

/ ********************************************************************************
函数功能:主程序
入口参数:
出口参数:
******************************************************************************** /
void main(void)
{
  DDRC =0xff;
  DDRA =0xff;

  DDRD =0xfb;
  PORTD =0x02;

  lcd_init();           // 初始化 LCD
  XY_String(0,0,"IR KEY Number:");
  while(1)
  {
    RecvRFStart();
    if(RF_Head = =1)
    {
      RecvRF();
  }
  switch(code3)
  {
  case 0x0C:          //1
        XY_Num(0,1,0);//显示高位
        XY_Num(1,1,1);//显示高位
        break;
  case 0x0D:           //2
        XY_Num(0,1,0);//显示高位
        XY_Num(1,1,2);//显示高位
        break;
  case 0x0E:          //3
        XY_Num(0,1,0);//显示高位
        XY_Num(1,1,3);//显示高位
        break;
```

```
  case 0x10:          //4
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,4);//显示高位
          break;
  case 0x11:          //5
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,5);//显示高位
          break;
  case 0x12:          //6
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,6);//显示高位
          break;
  case 0x14:          //7
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,7);//显示高位
          break;
  case 0x15:          //8
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,8);//显示高位
          break;
  case 0x16:          //9
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,9);//显示高位
          break;
  case 0x19:          //0
          XY_Num(0,1,0);//显示高位
          XY_Num(1,1,0);//显示高位
          break;
     }
   }
}
```

11.11　无线通信模块应用实例

通过无线的方式进行数据通信称为无线通信。在有些实际应用场合，现场环境不允许系统间的通信采用传统有线通信方式，此时数据交换就是只能以无线通信的方式。在日常生活中常见的无线通信有：手机和无绳电话、汽车遥控锁、遥控门等。根据我国国家无线电管理委员会分配给我国的陆地移动通信频率范围以及各种其他因素的综合考虑，真正适合当前作陆地移动通信的有 150MHz、230MHz（数据传输用）、350MHz、450MHz 和 900MHz 几个工

作频段，其中 350MHz 频段划归公安部门作公安专用，900MHz 作为 GSM 公用移动通信频段，因此，实际留给常规无线电台可用的频率资源非常有限。

11.11.1　无线通信模块原理与分类

无线通信的原理就是将数据加到载波上从而实现数据的传送。本节所介绍的无线收发模块包括两部分内容：编码/解码电路和高频电路。编码/解码电路主要负责对数据的编码/解码，高频部分负责将前端处理好的数据发送出去。无线通信模块按工作频段可以分为 350MHz 和 459MHz 等，按控制路数分可以分为单路、双路、三路……，无线接收模块按输出信号类型还可以分为锁存和非锁存。

11.11.2　无线通信模块主要技术指标

本节选用了市面上常见的无线收发模块来介绍单片机与无线收发模块间的软硬件接口。本实验选用了 YK200-4 型无线发送模块和带解码超再生接收模块，两者主要技术指标如下。



图 11-88　四键遥控发射器实物图

1. YK200-4（200m 桃木拨码四键遥控器）（见图 11-88）

型号规格：YJRF-YK200-4T

产品名称：200m 四键遥控器（桃木外壳）

发射/接收距离：200m

工作电压：DC 12V（电池供电）

尺寸：58mm × 39mm × 14mm

工作频率：315MHz、433MHz

工作电流：13mA

工作温度：－40 ~ ＋60℃

编码类型：固定码（板上焊盘跳接设置）

编码方式：　焊盘

应用说明：与各类型带解码功能的接收模块联合使用，解码输出后进行相应控制，如采用单片机进行读取接收并解码数据，然后控制相应的灯或电源开关。

2. 超再生解码接收板（见图 11-89）

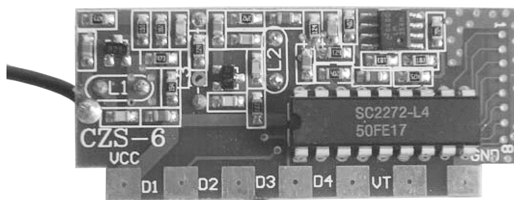


图 11-89　超再生解码接收板

型号规格：YJRF-JSM-Z

产品名称：带解码超再生接收模块（焊盘型）

工作电压：DC 5V

接收灵敏度：－103dBm

尺寸：49mm × 20mm × 7mm

工作频率：315MHz、433MHz

工作电流：5mA

工作温度：－40 ~ ＋60℃

编码类型：固定码（板上焊盘跳接设置）

产品说明：锁存（L4），非锁存（M4）

应用说明：与各类型遥控器配合使用，解码输出后进行相应控制，如采用单片机进行读取接收并解码数据，然后控制相应的灯或电源开关。

11.11.3　PT2262/PT2272 无线模块简介

目前，市场上出现了很多无线数据收发模块，如 PTR 2000、FB230 等。无线数据收发模块的性能优异，外围元器件少，设计、使用非常简单。无线收发模块一般在内部都集成了高频发射、高频接收、PLL 合成、FSK 调制/解调、参数放大、功率放大等功能。

在无线遥控领域，PT2262/PT2272 是目前最常用的芯片之一，本节就重点介绍这两个芯片的原理及相应模块的应用。无线收发模块的实物如图 11-90 所示。



图 11-90　无线收发模块实物图

1. PT2262/PT2272 工作原理

PT2262/PT2272 是我国台湾普城公司生产的一种 CMOS 工艺制造的低功耗低价位通用编码/解码电路，是目前在无线通信电路中作地址编码识别最常用的芯片之一。PT2262/PT2272 最多可有 12 位（A0 ~ A11）三态（悬空，接高电平，接低电平）地址设定引脚，任意组合可提供 531441 个地址码。PT2262 最多可有 6 位（D0 ~ D5）数据端引脚，设定的地址码和数据码从 17 脚（Dout）串行输出，可用于无线遥控发射电路。

PT2262 和 PT2272 的引脚排列如图 11-91 所示。对于编码器 PT2262，A0 ~ A5 共 6 根线为地址线，而 A6 ~ A11 共 6 根线可以作为地址线，也可以作为数据线，这要取决于所配合使用的解码器。若解码器没有数据线，则 A6 ~ A11 作为地址线使用，这种情况下，A0 ~ A11 共 12 根地址线，每线都可以设置成“1”、“0”、“开路”三种状态之一，因此共有编码数 $3^{12}=531441$ 种；但若配对使用的解码器的 A6 ~ A11 是数据线，例如 PT2272，那么这时 PT2262 的 A6 ~ A11 也作为数据线用，并只可设置为“1”和“0”两种状态之一，而地址线只剩下 A0 ~ A5 共 6 根，编码数降为 $3^{6}=729$ 种。

该编解码器的编码信号格式是：用 2 个周期的占空比为 1∶3（即高电平宽度为 1，低电平宽度为 2，周期为 3）的波形来表示 1 个“0”，用 2 个周期的占空比为 2∶3（即高电平宽度为 2，低电平宽度为 1，周期为 3）的波形来表示 1 个“1”，用 1 个周期的占空比为 1∶3 的波形紧跟着 1 个周期的占空比为 2∶3 的波形来表示“开路”。地址码和数据码都用宽度不同的脉冲来表示，两个窄脉冲表示“0”；两个宽脉冲表示“1”；1 个窄脉冲和 1 个宽脉冲

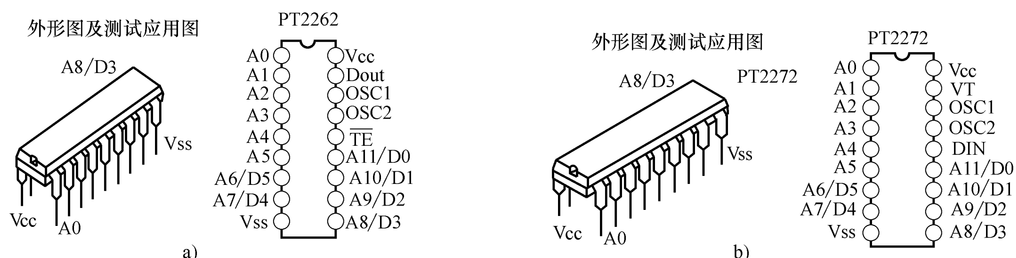


图 11-91　PT2262、PT2272 引脚排列图

表示“F”，也就是地址码的“悬空”。

编码芯片 PT2262 发出的编码信号由地址码、数据码、同步码组成一个完整的码字。解码芯片 PT2272 接收到信号后，其地址码经过两次比较核对后，VT 脚才输出高电平，与此同时相应的数据脚也输出高电平。PT2262 每次发射时至少发射 4 组字码，因为无线发射的特点，第 1 组字码非常容易受零电平干扰，往往会产生误码，所以 PT2272 只有在连续两次检测到相同的地址码和数据码时才会把数据码中的“1”驱动相应的数据输出端为高电平，驱动 VT 端同步为高电平。当发射机没有按键按下时，PT2262 不接通电源，其 17 脚为低电平，所以 315MHz 的高频发射电路不工作，当有按键按下时，PT2262 得电工作，其 17 脚输出经调制的串行数据信号，当 17 脚为高电平期间，315MHz 的高频发射电路起振并发射等幅高频信号，当 17 脚为低平期间，315MHz 的高频发射电路停止振荡，所以高频发射电路完全受控于 PT2262 的 17 脚输出的数字信号，从而对高频电路完成幅度键控（ASK 调制）相当于调制度为 100% 的调幅。

PT2272 解码芯片有不同的后缀，表示不同的功能，有 L4/M4/L6/M6 之分，其中 L 表示锁存输出，数据只要成功接收就能一直保持对应的电平状态，直到下次遥控数据发生变化时改变。M 表示非锁存输出，数据脚输出的电平是瞬时的，而且与发射端是否发射相对应，可以用于类似点动的控制。后缀的 6 和 4 表示有几路并行的控制通道，当采用 4 路并行数据时（PT2272-M4），对应的地址编码应该是 8 位，如果采用 6 路的并行数据时（PT2272-M6），对应的地址编码应该是 6 位。

PT2262 和 PT2272 除地址编码必须完全一致外，振荡电阻还必须匹配，一般要求译码器振荡频率要高于编码器振荡频率的 2.5 ~8 倍，否则接收距离会变近甚至无法接收，随着技术的发展市场上出现一批兼容芯片，在实际使用中只要对振荡电阻稍做改动就能配套使用。在具体的应用中，外接振荡电阻可根据需要进行适当的调节，阻值越大振荡频率越慢，编码的宽度越大，发送一帧的时间越长。市场上大部分产品都是用 PT2262/1.2MHz 和 PT2272/200kHz 组合的，少量产品用 PT2262/4.7MHz 和 PT2272/820kHz 组合。

PT2262 编码电路与 PT2272 解码电路一般配对使用，PT2262 的特点是在其内部已经把编码信号调制在一个较高的载频上。要把遥控编码信息用无线方式（红外线或无线电等）传送出去，必须有载体（载波），把编码信息“装载”在载体上（调制在载波上）才能传送出去，因此需要一个振荡电路和一个调制电路。PT2262 编码器内部，已包含了这些电路，从 DOUT 端送出的是调制好了的约 38kHz 的高频已调波，因此使用起来非常方便，适用于红外线和超声波遥控电路。PT2262 和 PT2272 引脚功能分别见表 11-21 和表 11-22。

表 11-21 编码电路 PT2262 引脚功能表

名称	引脚	说　明
D0 ~ D5	7 ~ 8、10 ~ 13	数据输入端,有一个为“1”即有编码发出,内部下拉
Vcc	18	电源正端(+)
Vss	9	电源负端(-)
TE	14	编码启动端,用于多数据的编码发射,低电平有效
OSC1	16	振荡电阻输入端,与 OSC2 所接电阻决定振荡频率
OSC2	15	振荡电阻振荡器输出端
Dout	17	编码输出端(正常时为低电平)

表 11-22 解码电路 PT2272 引脚功能表

名称	引脚	说　明
A0 ~ A11	1 ~ 8、10 ~ 13	地址引脚,用于进行地址编码,可置为“0”、“1”、“f”(悬空),必须与 PT2262 一致,否则不解码
D0 ~ D5	7 ~ 8、10 ~ 13	地址或数据引脚,当作为数据引脚时,只有在地址码与 PT2262 一致,数据引脚才能输出与 PT2262 数据端对应的高电平,否则输出为低电平,锁存型只有在接收到下一数据才能转换
Vcc	18	电源正端(+)
Vss	9	电源负端(-)
DIN	14	数据信号输入端,来自接收模块输出端
OSC1	16	振荡电阻输入端,与 OSC2 所接电阻决定振荡频率
OSC2	15	振荡电阻振荡器输出端
VT	17	解码有效确认输出端(常低),解码有效变成高电平(瞬态)

2. 基于 PT2262 的无线编码模块

编码发射模块外形小巧、美观，与很多车辆防盗系统中的遥控器一样。根据功能的多少按键数也不一样，本节所用的发射模块为 A、B、C、D 4 个按键。编码发射模块主要由 PT2262 编码集成电路和高频调制、功率放大电路组成，常用的编码发射模块实物和原理框图如图 11-92 所示。



图 11-92 编码发射模块实物图与原理框图

其中编码部分电路由 PT2262 编码集成电路来组成，具体电路如图 11-93 所示。

3. 基于 PT2272 的无线解码模块

解码接收模块包括接收头和解码芯片 PT2272 两部分组成。接收头将收到的信号输入 PT2272 的 14 脚（DIN），PT2272 再将收到的信号解码。解码接收模块和电路原理图如图 11-94 所示。

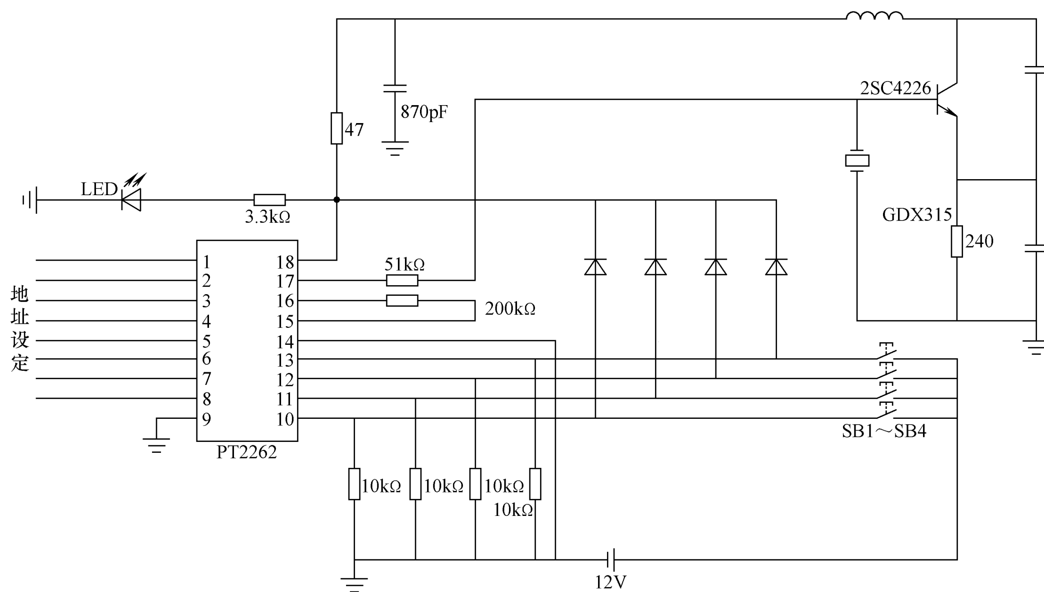
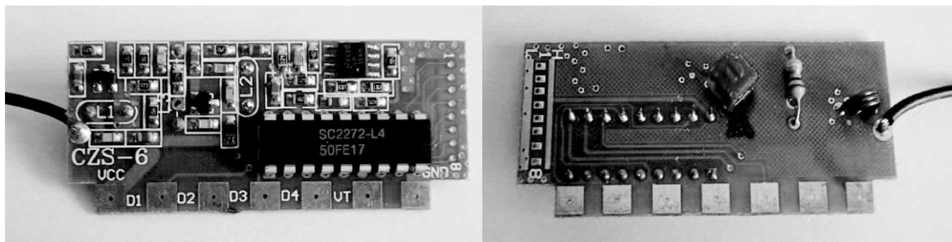


图 11-93　编码电路原理图



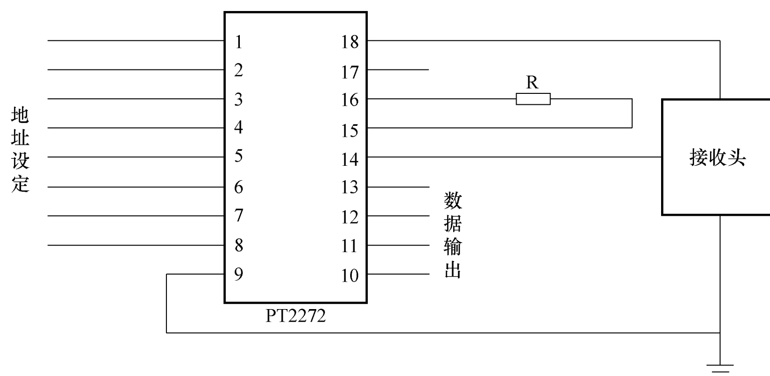


图 11-94　解码接收模块和电路原理图

4. 无线收发模块的地址码设定

在通常使用中，一般采用 8 位地址码和 4 位数据码，这时编码芯片 PT2262 和解码芯片 PT2272 的 1～8 脚为地址设定脚，有 3 种状态可供选择：悬空、接正电源、接地，地址编码不重复度为 $3^8=6561$ 组，只有发射端 PT2262 和接收端 PT2272 的地址编码完全相同，才能配对使用，遥控模块的生产厂商为了便于生产管理，出厂时遥控模块的 PT2262 和 PT2272 的 8 位地址编码端全部悬空，这样用户可以很方便地选择各种编码状态，用户如果想改变地

址编码，只要将 PT2262 和 PT2272 的 1 ~ 8 脚设置相同即可，例如将发射机的 PT2262 的 2 脚接地，3 脚接正电源，其他引脚悬空，那么接收机的 PT2272 只要 2 脚也接地，3 脚接正电源，其他引脚悬空就能实现配对接收。地址设置跳线如图 11-95 所示，用户可以在 PCB 上直接将地址引脚（PCB 中间 8 个过孔焊盘）与 L（低电平）或 H（高电平）相连，从而实现地址设置。PT2262 与 PT2272 地址设置要完全一样。当两者地址编码完全一致时，接收机对应的 D1 ~ D4 端输出约 4V 互锁高电平控制信号，同时 VT 端也输出解码有效高电平信号。

0 0 0 0 0 0 0 0　　L

_ _ _ _ _ _ _ _

1 1 1 1 1 1 1 1　　H

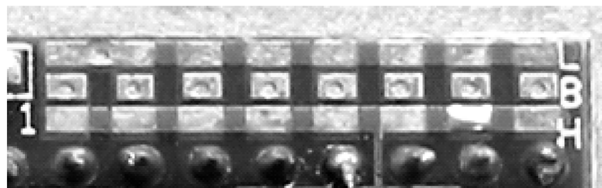


图 11-95　地址设置跳线图

11.11.4　无线通信模块的软硬件设计应用

在功能稍复杂的系统中仅靠一对无线收发模块往往达不到要求，很多情况下都要借助于单片机扩展出更多的功能。本例通过一个简单的例子，实现单片机与无线接收模块的组合应用。

在发射模块上按下 A、B、C、D 4 个键，接收模块将接收到的数据传送给单片机，在单片机上实现 LED 数码管显示。A、B、C、D 分别对应 1、2、3、4。即发射模块上按下 A 键，对应单片机接收到后在 LED 数码管上显示 0001，按下 B 键显示 0002……

1. 硬件原理图（见图 11-96）

2. 软件设计

```
/**********************************************************/
/* 无线模块测试程序                                        */
/* 目标器件:ATmega128                                      */
/* 晶振:RC 1MHz                                            */
/* 编译环境:ICCAVR 6.31A                                   */
/**********************************************************/

/*************************包含头文件*************************/
#include <iom16v.h>
#include <macros.h>

/*********************************************************
宏定义
**********************************************************/
char dat;                                    //接收到的数据
```

```
/************************数码管段码表************************/
extern const unsigned char
tab[ ] = { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,
                        /*   0    1    2    3    4    5    6    7    8    9   */
                              0x88,0x83,0xC6,0xA1,0x86,0x8E};
                        /*   a    b    c  d    e    f   */
/***********************************************************
函数功能:延时程序
入口参数:
出口参数:
***********************************************************/
void delay_1us(void)                    //1μs 延时函数
  {
  asm("nop");
  }

void delay_nus(unsigned int n)          //Nμs 延时函数
  {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1us();
  }

void delay_1ms(void)                    //1ms 延时函数
  {
    unsigned int i;
    for (i=0;i<140;i++);
  }

void delay_nms(unsigned int n)          //Nms 延时函数
  {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1ms();
  }

/***********************************************************
函数功能:显示子程序
入口参数:k
出口参数:
```

```
**********************************************************/
void display(unsigned int k)
{
    PORTC = tab[k/1000];
    PORTA = 0xef;
    delay_nms(2);

    PORTC = tab[k/100%10];
    PORTA = 0xf7;
    delay_nms(2);

    PORTC = tab[k/10%10];
    PORTA = 0xfb;
    delay_nms(2);

    PORTC = tab[k%10];
    PORTA = 0xfd;
    delay_nms(2);
}


/**********************************************************
函数功能:主程序
入口参数:
出口参数:
**********************************************************/
void main(void)
{
    unsigned char datavalue;
    DDRC = 0xff;
    PORTC = 0xff;
    DDRA = 0xff;
    PORTA = 0xff;
    DDRB = 0xe1;
    PORTB = 0xff;

while(1)
{
    dat = (PINB&0x1e);
    if(dat = = 0x02)//A
```

```
    datavalue =0x01;
    if(dat = =0x04)//B
    datavalue =0x02;
    if(dat = =0x10)//C
    datavalue =0x03;
    if(dat = =0x08)//d
    datavalue =0x04;
    display(datavalue);                //将读到的数显示
  }
}
```

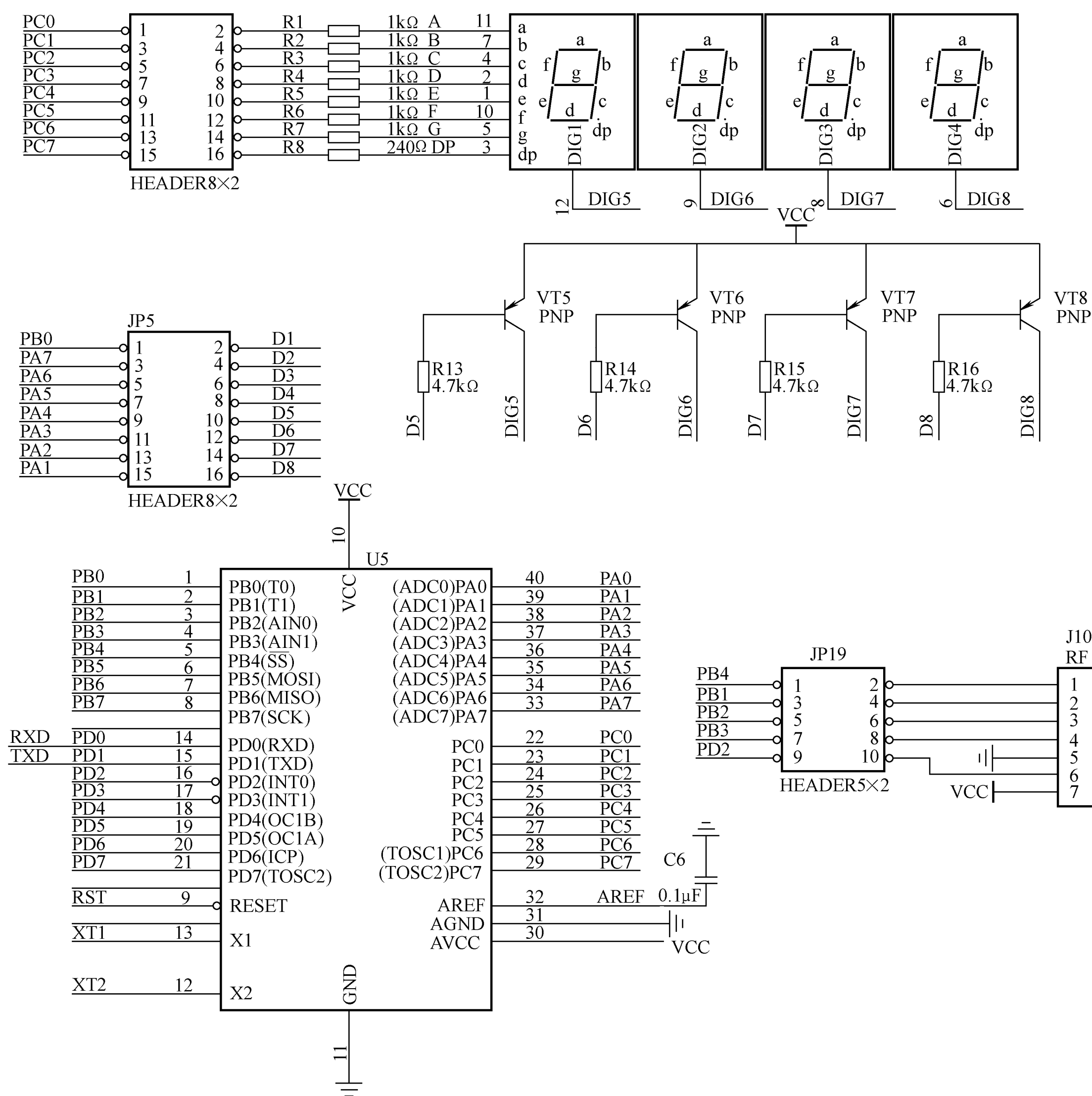


图 11-96　硬件原理图

11.12　PWM 应用实例

脉冲宽度调制（Pulse Width Modulation，PWM）简称脉宽调制。它是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用于测量、通信、功率控制与变换等许多领域。PWM 技术被广泛应用于电动机调速、D/A 转换、电源控制等领域。

11.12.1　PWM 的特点

模拟信号的值可以连续变化，其时间和幅度的分辨率都没有限制。9V 电池就是一种模拟器件，因为它的输出电压并不精确地等于 9V，而是随时间发生变化，并可取任何实数值。与此类似，从电池吸收的电流也不限定在一组可能的取值范围之内。模拟信号与数字信号的区别在于后者的取值通常只能属于预先确定的可能取值集合之内，例如在 {0V，5V} 这一集合中取值。

模拟电压和电流可直接用来进行控制，如对汽车收音机的音量进行控制。在简单的模拟收音机中，音量旋钮被连接到一个可变电阻。拧动旋钮时，电阻值变大或变小；流经这个电阻的电流也随之增加或减少，从而改变了驱动扬声器的电流值，使音量相应变大或变小。与收音机一样，模拟电路的输出与输入成线性比例。

尽管模拟控制看起来可能直观而简单，但它并不总是非常经济或可行的。其中一点就是，模拟电路容易随时间漂移，因而难以调节。能够解决这个问题的精密模拟电路可能非常庞大、笨重（如老式的家庭立体声设备）和昂贵。模拟电路还有可能严重发热，其功耗相对于工作元件两端电压与电流的乘积成正比。模拟电路还可能对噪声很敏感，任何扰动或噪声都肯定会改变电流值的大小。

通过以数字方式控制模拟电路，可以大幅度降低系统的成本和功耗。此外，许多微控制器和 DSP 已经在芯片上包含了 PWM 控制器，这使数字控制的实现变得更加容易了。

简而言之，PWM 是一种对模拟信号电平进行数字编码的方法。通过高分辨率计数器的使用，方波的占空比被调制用来对一个具体模拟信号的电平进行编码。PWM 信号仍然是数字的，因为在给定的任何时刻，满幅值的直流供电要么完全有（ON），要么完全无（OFF）。电压或电流源是以一种通（ON）或断（OFF）的重复脉冲序列被加到模拟负载上去的。通的时候即是直流供电被加到负载上的时候，断的时候即是供电被断开的时候。只要带宽足够，任何模拟值都可以使用 PWM 进行编码。

图 11-97 显示了 3 种不同的 PWM 信号。图 a 是一个占空比为 10% 的 PWM 输出，即在信号周期中，10% 的时间通，其余 90% 的时间断。图 b 和图 c 显示的分别是占空比为 50% 和 90% 的 PWM 输出。这 3 种 PWM 输出编码的分别是强度为满度值的 10%、50% 和 90% 的 3 种不同模拟信号值。例如，假设供电电源为 9V，占空比为 10%，则对应的是一个幅度为 0.9V 的模拟信号。

图 11-98 是一个可以使用 PWM 进行驱动的简单电路。图中使用 9V 电池来给一个白炽灯泡供电。如果将连接电池和灯泡的开关闭合 50ms，灯泡在这段时间中将得到 9V 供电。如果在下一个 50ms 中将开关断开，灯泡得到的供电将为 0V。如果在 1s 内将此过程重复 10 次，灯泡将会点亮并像连接到了一个 4.5V 电池（9V 的 50%）上一样。这种情况下，占空比为 50%，调制频率为 10Hz。

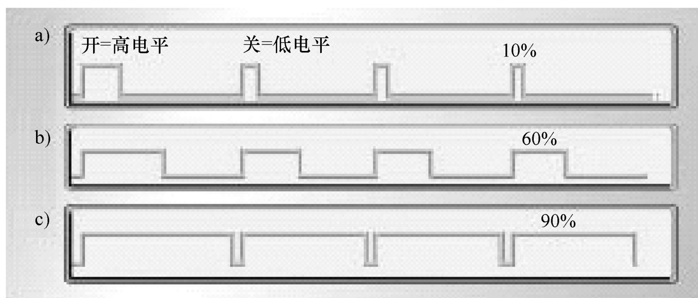


图 11-97　3 种不同的 PWM 波形



图 11-98　PWM 驱动示意图

大多数负载（无论是电感性负载还是电容性负载）需要的调制频率高于 10Hz。设想一下如果灯泡先接通 5s 再断开 5s，然后再接通、再断开……占空比仍然是 50%，但灯泡在前 5s 内将点亮，在下一个 5s 内将熄灭。要让灯泡取得 4.5V 电压的供电效果，通断循环周期与负载对开关状态变化的响应时间相比必须足够短。要想取得调光灯（但保持点亮）的效果，必须提高调制频率。在其他 PWM 应用场合也有同样的要求。通常调制频率为 1 ~ 200kHz 之间。

PWM 的一个优点是从处理器到被控系统信号都是数字形式的，无需进行数模转换。让信号保持为数字形式可将噪声影响降到最小。噪声只有在强到足以将逻辑 1 改变为逻辑 0 或将逻辑 0 改变为逻辑 1 时，也才能对数字信号产生影响。

对噪声抵抗能力的增强是 PWM 相对于模拟控制的另外一个优点，而且这也是在某些时候将 PWM 用于通信的主要原因。从模拟信号转向 PWM 可以极大地延长通信距离。在接收端，通过适当的 RC 或 LC 网络可以滤除调制高频方波，并将信号还原为模拟形式。

11.12.2　ATmega16 内部 PWM 简介

ATmega16 内部带有 4 路独立 PWM 功能。8 位计数/定时器 0、2 以及 16 位计数/定时器 1 都具备 PWM 功能。

1. 相关寄存器

Atmega16 的定时/计数器 0、1、2 都具有 PWM 功能，所以在使用 PWM 功能时要先掌握各寄存器的配置。在定时/计数器 0 中主要有 T/C 寄存器 TCNT0、输出比较寄存器 OCR0、T/C 中断屏蔽寄存器 TIMSK 和 T/C 中断标志寄存器 TIFR，下面简单介绍一下这 4 个寄存器的功能。

通过 T/C 寄存器 TCNT0 可以直接对计数器的 8 位数据进行读写访问。对 TCNT0 寄存器的写访问将在下一个时钟阻止比较匹配。在计数器运行的过程中修改 TCNT0 的数值有可能丢失一次 TCNT0 和 OCR0 的比较匹配。

（1）T/C 寄存器 TCNT0

bit	7	6	5	4	3	2	1	0
	TCNT0[7:0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

（2）输出比较寄存器 OCR0

bit	7	6	5	4	3	2	1	0
	OCR0[7:0]							
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

输出比较寄存器包含一个 8 位的数据，不间断地与计数器数值 TCNT0 进行比较。匹配事件可以用来产生输出比较中断，或者用来在 OCR0 引脚上产生波形。

（3）T/C 中断屏蔽寄存器 TIMSK

bit	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

· bit 7-OCIE2：T/C2 输出比较匹配中断使能

当 OCIE2 和状态寄存器的全局中断使能位 I 都为“1”时，T/C2 的输出比较匹配 A 中断使能。当 T/C2 的比较匹配发生，即 TIFR 中的 OCF2 置位时，中断服务程序得以执行。

· bit 6-TOIE2：T/C2 溢出中断使能

当 TOIE2 和状态寄存器的全局中断使能位 I 都为“1”时，T/C2 的溢出中断使能。当 T/C2 发生溢出，即 TIFR 中的 TOV2 置位时，中断服务程序得以执行。

· bit 5-TICIE1：T/C1 输入捕捉中断使能

当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的输入捕捉中断使能。一旦 TIFR 的 ICF1 置位，CPU 即开始执行 T/C1 输入捕捉中断服务程序。

· bit 4-OCIE1A：输出比较 A 匹配中断使能

当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的输出比较匹配 A 中断使能。一旦 TIFR 上的 OCF1A 置位，CPU 即开始执行 T/C1 输出比较 A 匹配中断服务程序。

· bit 3-OCIE1B：T/C1 输出比较匹配 B 中断使能

当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，使能 T/C1 的输出比较匹配 B 中断使能。一旦 TIFR 上的 OCF1B 置位，CPU 即开始执行 T/C1 输出比较匹配 B 中断服务程序。

· bit 2-TOIE1：T/C1 溢出中断使能

当该位被设为“1”，且状态寄存器中的 I 位被设为“1”时，T/C1 的溢出中断使能。一旦 TIFR 上的 TOV1 置位，CPU 即开始执行 T/C1 溢出中断服务程序。

· bit 1-OCIE0：T/C0 输出比较匹配中断使能

当 OCIE0 和状态寄存器的全局中断使能位 I 都为“1”时，T/C0 的输出比较匹配中断使能。当 T/C0 的比较匹配发生，即 TIFR 中的 OCF0 置位时，中断服务程序得以执行。

· bit 0-TOIE0：T/C0 溢出中断使能

当 TOIE0 和状态寄存器的全局中断使能位 I 都为“1”时，T/C0 的溢出中断使能。当 T/C0 发生溢出，即 TIFR 中的 TOV0 置位时，中断服务程序得以执行。

（4）T/C 中断标志寄存器 TIFR

bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

· bit 7-OCF2：输出比较标志 2

当 T/C2 与 OCR2（输出比较寄存器 2）的值匹配时，OCF2 置位。此位在中断服务程序里硬件清零，也可以通过对其写 1 来清零。当 SREG 中的位 I、OCIE2 和 OCF2 都置位时，中断服务程序得到执行。

· bit 6-TOV2：T/C2 溢出标志

当 T/C2 溢出时，TOV2 置位。执行相应的中断服务程序时此位硬件清零。此外，TOV2 也可以通过写 1 来清零。当 SREG 中的位 I、TOIE2 和 TOV2 都置位时，中断服务程序得到执行。在 PWM 模式中，当 T/C2 在 0x00 改变记数方向时，TOV2 置位。

· bit 5-ICF1：T/C1 输入捕捉标志位

外部引脚 ICP1 出现捕捉事件时 ICF1 置位。此外，当 ICR1 作为计数器的 TOP 值时，一旦计数器值达到 TOP，ICF1 也置位。执行输入捕捉中断服务程序时 ICF1 自动清零。也可以对其写入逻辑“1”来清除该标志位。

· bit 4-OCF1A：T/C1 输出比较匹配 A 标志位

当 TCNT1 与 OCR1A 匹配成功时，该位被设为“1”。强制输出比较（FOC1A）不会置位 OCF1A。执行强制输出比较匹配 A 中断服务程序时 OCF1A 自动清零。也可以对其写入逻辑“1”来清除该标志位。

· bit 3-OCF1B：T/C1 输出比较匹配 B 标志位

当 TCNT1 与 OCR1B 匹配成功时，该位被设为“1”。强制输出比较（FOC1B）不会置位 OCF1B。执行强制输出比较匹配 B 中断服务程序时 OCF1B 自动清零。也可以对其写入逻辑“1”来清除该标志位。

· bit 2-TOV1：T/C1 溢出标志

该位的设置与 T/C1 的工作方式有关。工作于普通模式和 CTC 模式时，T/C1 溢出时 TOV1 置位。对工作在其他模式下的 TOV1 标志位置位。执行溢出中断服务程序时 OCF1A 自动清零。也可以对其写入逻辑“1”来清除该标志位。

· bit 1-OCF0：输出比较标志 0

当 T/C0 与 OCR0（输出比较寄存器 0）的值匹配时，OCF0 置位。此位在中断服务程序里硬件清零，也可以对其写 1 来清零。当 SREG 中的位 I、OCIE0（T/C0 比较匹配中断使

能）和 OCF0 都置位时，中断服务程序得到执行。

· bit 0-TOV0：T/C0 溢出标志

当 T/C0 溢出时，TOV0 置位。执行相应的中断服务程序时此位硬件清零。此外，TOV0 也可以通过写 1 来清零。当 SREG 中的位 I、TOIE0（T/C0 溢出中断使能）和 TOV0 都置位时，中断服务程序得到执行。在相位修正 PWM 模式中，当 T/C0 在 0x00 改变记数方向时，TOV0 置位。在定时/计数器 1 中的寄存器与在定时/计数器 0 中有所不同，因为在定时/计数器 1 为 16 位定时/计数器，主要区别是 T/C1 控制寄存器 ATCCR1A、T/C1 控制寄存器 BTCCR1B、T/C 寄存器 TCNT1H 与 TCNT1L、输出比较寄存器 1AOCR1AH 与 OCR1AL、输出比较寄存器 1BOCR1BH 与 OCR1BL 以及输入捕捉寄存器 1ICR1H 与 ICR1L。另外几个寄存器 T/C1 中断屏蔽寄存器 TIMSK 和 T/C 中断标志寄存器 TIFR 在此不作重复介绍。

（5）T/C1 控制寄存器 ATCCR1A

bit	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

· bit 7:6-COM1A1:0：通道 A 的比较输出模式

· bit 5:4-COM1B1:0：通道 B 的比较输出模式

COM1A1:0 与 COM1B1:0 分别控制 OC1A 与 OC1B 状态。如果 COM1A1:0（COM1B1:0）的一位或两位被写入“1”，OC1A（OC1B）输出功能将取代 I/O 端口功能。此时 OC1A（OC1B）相应的输出引脚数据方向控制必须置位以使能输出驱动器。

OC1A（OC1B）与物理引脚相连时，COM1x1:0 的功能由 WGM13:0 的设置决定。下面给出当 WGM13:0 在各种工作模式下的功能定义。

●比较输出模式，非 PWM

COM1A1/COM1B1	COM1A0/COM1B0	说　明
0	0	普通端口操作,非 OCR1A/OCR1B 功能
0	1	比较匹配时 OCR1A/OCR1B 电平取反
1	0	比较匹配时清零 OCR1A/OCR1B(输出低电平)
1	1	比较匹配时置位 OCR1A/OCR1B（输出高电平）

●比较输出模式，快速 PWM

COM1A1/COM1B1	COM1A0/COM1B0	说　明
0	0	普通端口操作,非 OCR1A/OCR1B 功能
0	1	WGM13:0 = 15,比较匹配时 OC1A 取反,OC1B 不占用物理引脚。WGM13:0 为其他值时为普通端口操作,非 OC1A/OC1B 功能
1	0	比较匹配时清零 OC1A/OC1B,OC1A/OC1B 在 TOP 时置位
1	1	比较匹配时置位 OC1A/OC1B,OC1A/OC1B 在 TOP 时清零

●比较输出模式，相位修正及相频修正 PWM 模式

COM1A1/COM1B1	COM1A0/COM1B0	说　明
0	0	普通端口操作，非 OCR1A/OCR1B 功能
0	1	WGM13:0 =9 或 14，比较匹配时 OC1A 取反，OC1B 不占用物理引脚。WGM13:0 为其他值时为普通端口操作，非 OC1A/OC1B 功能
1	0	升序记数时比较匹配将清零 OC1A/OC1B，降序记数时比较匹配将置位 OC1A/OC1B
1	1	升序记数时比较匹配将置位 OC1A/OC1B，降序记数时比较匹配将清零 OC1A/OC1B

· bit 3-FOC1A：通道 A 强制输出比较

· bit 2-FOC1B：通道 B 强制输出比较

FOC1A/FOC1B 只有当 WGM13:0 指定为非 PWM 模式时被激活。为与未来器件兼容，工作在 PWM 模式下对 TCCR1A 写入时，这两位必须清零。当 FOC1A/FOC1B 位置 1，立即强制波形产生单元进行比较匹配。COM1x1:0 的设置改变 OC1A/OC1B 的输出。注意 FOC1A/FOC1B 位作为选通信号。COM1x1:0 位的值决定强制比较的效果。

在 CTC 模式下使用 OCR1A 作为 TOP 值，FOC1A/FOC1B 选通既不会产生中断也不好清除定时器。

FOC1A/FOC1B 位总是读为 0。

· bit 1:0-WGM11:0：波形发生模式

这两位与位于 TCCR1B 寄存器的 WGM13:2 相结合，用于控制计数器的计数序列——计数器计数的上限值和确定波形发生器的工作模式（ 见表 11-23）。T/C 支持的工作模式有：普通模式（ 计数器），比较匹配时清零定时器（CTC）模式，及 3 种脉宽调制（PWM）模式。

表 11-23　波形发生模式

模式	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	定时器/计数器工作模式	TOP	OCR1x 更新时刻	TOV1 置位时刻
0	0	0	0	0	普通模式	0xFFFF	立即更新	MAX
1	0	0	0	1	8 位相位修正 PWM	0x00FF	TOP	BOTTOM
2	0	0	1	0	9 位相位修正 PWM	0x01FF	TOP	BOTTOM
3	0	0	1	1	10 位相位修正 PWM	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	立即更新	MAX
5	0	1	0	1	8 位快速 PWM	0x00FF	TOP	TOP
6	0	1	1	0	9 位快速 PWM	0x01FF	TOP	TOP
7	0	1	1	1	10 位快速 PWM	0x03FF	TOP	TOP
8	1	0	0	0	相位与频率修正 PWM	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	相位与频率修正 PWM	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	相位修正 PWM	ICR1	TOP	BOTTOM
11	1	0	1	1	相位修正 PWM	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	立即更新	MAX
13	1	1	0	1	保留	—	—	—
14	1	1	1	0	快速 PWM	ICR1	TOP	TOP
15	1	1	1	1	快速 PWM	OCR1A	TOP	TOP

（6）T/C1 控制寄存器 BTCCR1B

bit	7	6	5	4	3	2	1	0
	ICNC1	ICES1	—	WGM13	WGM12	CS12	CS11	CS10
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
初值	0	0	0	0	0	0	0	0

· bit 7-ICNC1：输入捕捉噪声抑制器

置位 ICNC1 将使能输入捕捉噪声抑制功能。此时外部引脚 ICP1 的输入被滤波。其作用是从 ICP1 引脚连续进行 4 次采样。如果 4 个采样值都相等，那么信号送入边沿检测器。因此使能该功能使得输入捕捉被延迟了 4 个时钟周期。

· bit 6-ICES1：输入捕捉触发沿选择

该位选择使用 ICP1 上的哪个边沿触发捕获事件。ICES1 为“0" 选择的是下降沿触发输入捕捉；ICES1 为“1" 选择的是逻辑电平的上升沿触发输入捕捉。按照 ICES1 的设置捕获到一个事件后，计数器的数值被复制到 ICR1 寄存器。捕获事件还会置为 ICF1。如果此时中断使能，输入捕捉事件即被触发。当 ICR1 用作 TOP 值（见 TCCR1A 与 TCCR1B 寄存器中 WGMn3：0 位的描述）时，ICP1 与输入捕捉功能脱开，从而输入捕捉功能被禁用。

· bit 5-保留位

该位保留。为保证与将来器件的兼容性，写 TCCR1B 时，该位必须写入“0"。

· bit 4:3-WGM13:2：波形发生模式

见 TCCR1A 寄存器中的描述。

· bit 2:0-CSn2:0：时钟选择

这 3 位用于选择 T/C1 的时钟源，见表 11-24。

表 11-24　T/C1 的时钟源选择位

CSn2	CSn1	CSn0	说　　明
0	0	0	无时钟源(T/C 停止)
0	0	1	$clk_{I/O}$/1(无预分频器)
0	1	0	$clk_{I/O}$/8(来自预分频器)
0	1	1	$clk_{I/O}$/64(来自预分频器)
1	0	0	$clk_{I/O}$/256(来自预分频器)
1	0	1	$clk_{I/O}$/1024(来自预分频器)
1	1	0	外部 Tn 引脚,下降沿驱动
1	1	1	外部 Tn 引脚,上升沿驱动

选择使用外部时钟源后，即使 T1 引脚被定义为输出，其 1 引脚上的逻辑信号电平变化仍然会驱动 T/C1 计数，这个特性允许用户通过软件来控制计数。

（7）T/C 寄存器 TCNT1H 与 TCNT1L

bit	7	6	5	4	3	2	1	0	
	TCNT1[15:8]								TCNT1H
	TCNT1[7:0]								TCNT1L
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初值	0	0	0	0	0	0	0	0	

TCNT1H 与 TCNT1L 组成了 T/C1 的数据寄存器 TCNT1。通过它们可以直接对定时器/计数器单元的 16 位计数器进行读写访问。为保证 CPU 对高字节与低字节的同时读写，必须使用一个 8 位临时高字节寄存器 TEMP。TEMP 是所有的 16 位寄存器共用的。

（8）输出比较寄存器 1AOCR1AH 与 OCR1AL

bit	7	6	5	4	3	2	1	0	
	OCR1A[15:8]								OCR1AH
	OCR1A[7:0]								OCR1AL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初值	0	0	0	0	0	0	0	0	

（9）输出比较寄存器 1BOCR1BH 与 OCR1BL

bit	7	6	5	4	3	2	1	0	
	OCR1B[15:8]								OCR1BH
	OCR1B[7:0]								OCR1BL
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初值	0	0	0	0	0	0	0	0	

该寄存器中的 16 位数据与 TCNT1 寄存器中的计数值进行连续的比较，一旦数据匹配，将产生一个输出比较中断，或改变 OC1x 的输出逻辑电平。

输出比较寄存器长度为 16 位。为保证 CPU 对高字节与低字节的同时读写，必须使用一个 8 位临时高字节寄存器 TEMP。TEMP 是所有的 16 位寄存器共用的。

（10）输入捕捉寄存器 1ICR1H 与 ICR1L

bit	7	6	5	4	3	2	1	0	
	ICR1[15:8]								ICR1H
	ICR1[7:0]								ICR1L
读/写	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
初值	0	0	0	0	0	0	0	0	

当外部引脚 ICP1（或 T/C1 的模拟比较器）有输入捕捉触发信号产生时，计数器 TCNT1 中的值写入 ICR1 中。ICR1 的设定值可作为计数器的 TOP 值。

输入捕捉寄存器长度为 16 位。为保证 CPU 对高字节与低字节的同时读写，必须使用一个 8 位临时高字节寄存器 TEMP。TEMP 是所有的 16 位寄存器共用的。

2. PWM 的工作模式

ATmega16 的 PWM 主要有 4 种工作模式：CTC 模式、快速 PWM 模式、相位修正 PWM 模式以及相位和频率修正 PWM 模式。下面以计数/定时器 1 为例简单介绍一下这 4 种 PWM 模式。

1）CTC 模式：在 CTC 模式（WGM13:0 =4 或 12）中 OCR1A 或 ICR1 寄存器用于调节计数器的分辨率。当计数器的数值 TCNT1 等于 OCR1A（WGM13:0 =4）或 ICR1（WGM13:0 =12）时计

数器清零。OCR1A 或 ICR1 定义了计数器的 TOP 值，即计数器的分辨率。波形发生器能够产生的最大频率为 $f_{OC2}=f_{clk_I/O}/2$（OCR1A = 0x0000）。频率由如下公式确定：

$$f_{OCnA}=\frac{f_{clk_I/O}}{2N(1+OCRnA)}$$

2）快速 PWM 模式：快速 PWM 模式与其他 PWM 模式的不同之处是其单边斜坡工作方式。计数器从 BOTTOM 计到 TOP，然后立即回到 BOTTOM 重新开始。对于正向的比较输出模式，输出比较引脚 OC1x 在 TCNT1 与 OCR1x 匹配时置位，在 TOP 时清零；对于反向比较输出模式，OCR1x 的动作正好相反。PWM 分辨率位数可用下式计算：

$$f_{OCnxPWM}=\frac{f_{clk_I/O}}{N(1+TOP)}$$

3）相位修正 PWM 模式：与相位和频率修正模式类似，此模式基于双斜坡操作。计时器重复地从 BOTTOM 计到 TOP，然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出模式下，当计时器向 TOP 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将清零为低电平；而在计时器向 BOTTOM 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将置位为高电平。工作于反向比较输出时则正好相反。工作于相位修正模式时 PWM 频率可由如下公式获得：

$$f_{OCnxPCPWM}=\frac{f_{clk_I/O}}{2NTOP}$$

4）相位和频率修正 PWM 模式：与相位修正模式类似，相频修正 PWM 模式基于双斜坡操作。计时器重复地从 BOTTOM 计到 TOP，然后又从 TOP 倒退回到 BOTTOM。在一般的比较输出模式下，当计时器向 TOP 计数时，若 TCNT1 与 OCR1x 匹配，OC1x 将清零为低电平；而在计时器向 BOTTOM 计数时，TCNT1 与 OCR1x 匹配，OC1x 将置位为高电平。输出的 PWM 频率可以通过如下公式计算得到：

$$f_{OCnxPFCPWM}=\frac{f_{clk_I/O}}{2NTOP}$$

11.12.3　基于 ATmega16 的 PWM 应用设计

利用 ATmega16 的 PWM 功能可以很方便地产生各种模拟波形，下面的例子利用内部 3 个定时器的快速 PWM 模式产生 50% 占空比的 PWM 波形。快速 PWM 的基本原理：计数器从 BOTTOM（0）计到 MAX（0xFF），然后立即回到 BOTTOM（0）重新开始。对于正向（COM01:COM00 = 10）比较输出模式，输出比较引脚 OC0 在 TCNT0 与比较匹配寄存器 OCR0 匹配时清零，在 BOTTOM（0）时置位；对于反向比较输出模式（COM01:COM00 = 11），OC0 的动作正好相反，如图 11-99 所示。

1. PWM 电路硬件原理图

对于 ATmega16 来说，仅靠自身电路就可产生 PWM 波形，硬件原理图如图 11-100 所示。

2. 软件设计

本例利用 ATmega16 的 PWM 模块产生 4 路 PWM 波，系统时钟为 1MHz，分频系数为 8，产生的 PWM 波频率约为 488Hz（参考快速 PWM 频率计算公式）。

下面是本例参考程序：

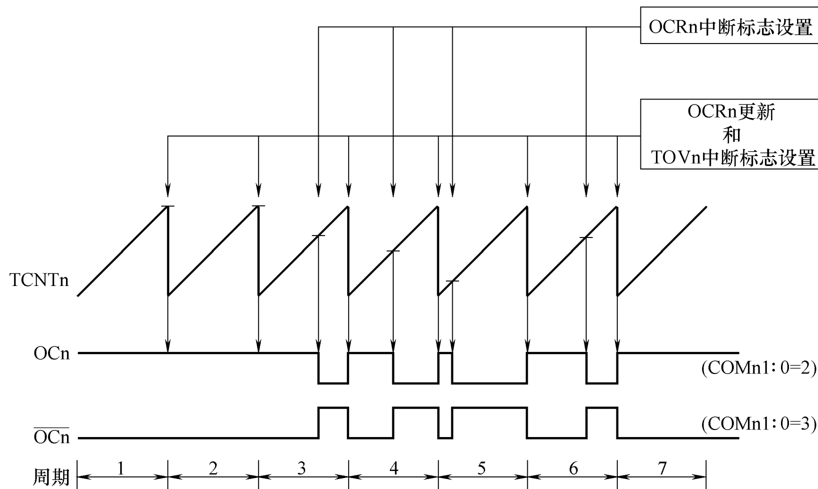


图 11-99　快速 PWM 工作时序

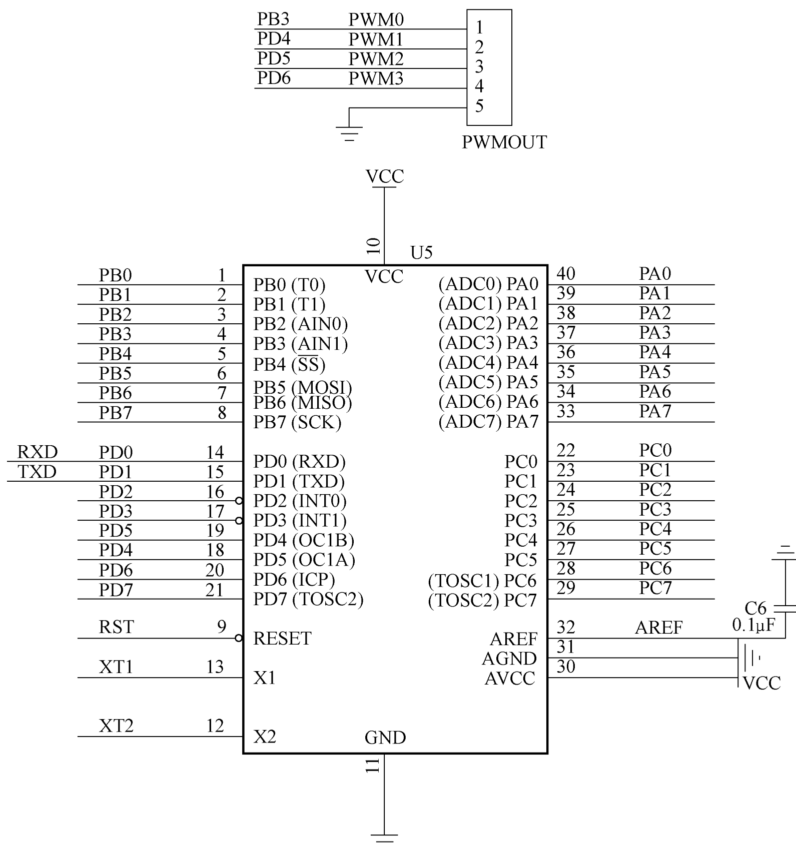


图 11-100　PWM 输出滤波电路

```
/ ************************************************************ /
/ *  PWM 硬件测试程序                                          * /
/ * 目标器件:ATmega16                                          * /
```

```
/ * 晶振:RC 1MHz                                                    * /
/ * 编译环境:ICCAVR 6. 31A                                           * /
/ ****************************************************************** /

/*************************包含头文件**************************/
#include  < iom16v. h >
#include  < macros. h >
/**************************宏定义**************************/
#define uchar unsigned char
#define uint unsigned int
/ ******************************************************************
函数功能:PWM0 初始化程序
入口参数:
出口参数:
 ****************************************************************** /
void pwm0_init( void)
{
    DDRB| =0b00001000;           //PB3 输出
    TCCR0 =0x00;
    OCR0 =0x7f;                  //50%
    TCNT0 =0;
    TCCR0 =0b01101010;           //快速 PWM 8 分频
}

/ ******************************************************************
函数功能:PWM1 初始化程序
入口参数:
出口参数:
 ****************************************************************** /
void pwm1_init( )
{
    DDRD| =0b00110000;
    TCCR1A| =0b10100010;
    TCCR1B| =0b00011001;            //快速 PWM 8 分频
    ICR1 =0x7ff;
    OCR1A =0x400;                   //PD5,50%
    OCR1B =0x400;                   //PD4
}

/ ******************************************************************
函数功能:PWM2 初始化程序
```

```
入口参数:
出口参数:
*********************************************************/
void pwm2_init()
{
    DDRD| =0b10000000;
    TCCR2| =0x00;
    OCR2 =0x7f;
    TCNT2 =0x00;                //PD7
    TCCR2| =0b01101010;         //快速 PWM 8 分频
}
/*********************************************************
函数功能:主程序
入口参数:
出口参数:
*********************************************************/
void main()
{
    char temp;
    pwm0_init();
    pwm1_init();
    pwm2_init();
    TIMSK =0xDA;
    while(1)
    {  }
}
```

11.13　SD 卡读写实例

SD 卡在现在的日常生活与工作中使用非常广泛，时下已经成为最为通用的数据存储卡。在诸如 MP3、数码相机等设备上也都采用 SD 卡作为其存储设备。SD 卡之所以得到如此广泛的使用，是因为它有价格低廉、存储容量大、使用方便、通用性与安全性强等优点。

既然它有着这么多优点，那么如果将它加入到单片机应用开发系统中来，将使系统变得更加出色。这就要求对 SD 卡的硬件与读写时序进行研究。对于 SD 卡的硬件结构，在官方的文档上有很详细的介绍，如 SD 卡内的存储器结构、存储单元组织方式等内容。要实现对它的读写，最核心的是它的时序，下文将对 SD 卡的读写进行简单介绍。

11.13.1　SD 卡简介

1. SD 卡的引脚定义

SD 卡外观尺寸相信读者都已非常熟悉，一般在数码相机、MP3 等多媒体设计中都可以看到。SD 卡引脚如图 11-101 所示。

从 SD 卡的引脚图上可以看出，其带有 9 个引脚，这些引脚是以金属片的形式供用户使用，在使用中一般会用配套的 SD 卡座来将引脚转接，方便引线。9 个引脚功能见表 11-25。

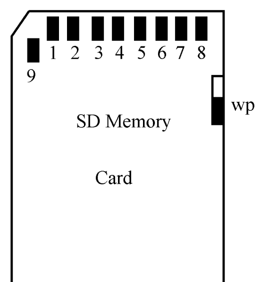


图 11-101　SD 卡引脚图

1）S：电源；I：输入；O：输出（推挽式）；PP：输入/输出使用推挽式驱动；

2）所有的 DAT 线（DAT1 ~ DAT3）在上电后只做输入。在 SET_ BUS_ WIDTH 命令之后，开始作为数据线。

3）在通电之后，信号线有 50kΩ 上拉电阻（可以作为卡检验或 SPI 模态选择），在普通数据传输期间，用户可以使用 SET_ CLR_ CARD_ DETECT（ACMD42）命令断开上拉。

表 11-25　SD 卡引脚功能

引脚编号	SD 模式			SPI 模式		
	名称	类型	描述	名称	类型	描述
1	CD/DAT3	IO 或 PP	卡检测/数据线 3	#CS	I	片选
2	CMD	PP	命令/回应	DI	I	数据输入
3	VSS1	S	电源地	VSS	S	电源地
4	VDD	S	电源	VDD	S	电源
5	CLK	I	时钟	SCLK	I	时钟
6	VSS2	S	电源地	VSS2	S	电源地
7	DAT0	IO 或 PP	数据线 0	DO	O 或 PP	数据输出
8	DAT1	IO 或 PP	数据线 1	RSV		
9	DAT2	IO 或 PP	数据线 2	RSV		

2. 用 SPI 方式驱动 SD 卡的方法

SD 卡支持两种总线方式：SD 方式与 SPI 方式。其中 SD 方式采用 6 线制，使用 CLK、CMD、DAT0 ~ DAT3 进行数据通信。而 SPI 方式采用 4 线制，使用 CS、CLK、DI、DO 进行数据通信。SD 方式的数据传输速度比 SPI 方式要快，采用单片机对 SD 卡进行读写时一般都采用 SPI 模式。采用不同的初始化方式可以使 SD 卡工作于 SD 方式或 SPI 方式。在普通 8 位单片机控制系统中只对其 SPI 方式进行介绍。

SD 卡的 SPI 通信接口使其可以通过 SPI 通道进行数据读写。从应用的角度来看，采用 SPI 接口的好处在于，很多单片机内部自带 SPI 控制器，不仅给开发上带来方便，同时也降低了开发成本。然而，它也有不好的地方，如失去了 SD 卡的性能优势，要解决这一问题，就要用 SD 方式，因为它提供更大的总线数据带宽。SPI 接口的选用是在上电初始向其写入第 1 个命令时进行的。

3. SD 卡命令与数据传输

（1）SD 卡的命令传输

SD 卡自身有完备的命令系统，以实现各项操作。命令格式如图 11-102 所示。

命令的传输过程采用发送应答机制，过程如图 11-103 所示。

每一个命令都有自己的命令应答格式。在 SPI 模式中定义了 3 种应答格式，见表 11-26 ~ 表 11-28。

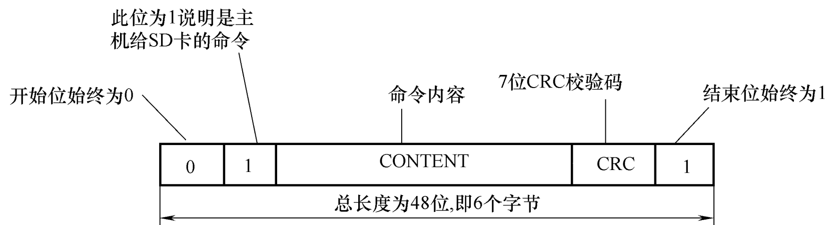


图 11-102　SD 卡的命令格式

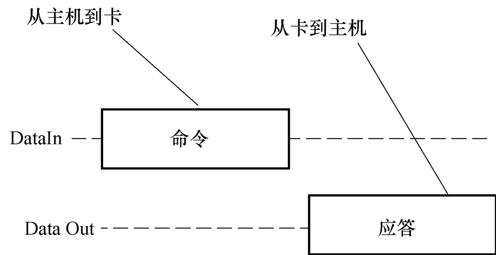


图 11-103　命令的传输过程

表 11-26　第 1 种格式

字　　节	位	含　　义
1	7	开始位,始终为 0
	6	参数错误
	5	地址错误
	4	擦除序列错误
	3	CRC 错误
	2	非法命令
	1	擦除复位
	0	闲置状态

表 11-27　第 2 种格式

字　　节	位	含　　义
1	7	开始位,始终为 0
	6	参数错误
	5	地址错误
	4	擦除序列错误
	3	CRC 错误
	2	非法命令
	1	擦除复位
	0	闲置状态
2	7	溢出,CSD 覆盖
	6	擦除参数
	5	写保护非法

（续）

字　节	位	含　义
2	4	卡 ECC 失败
	3	卡控制器错误
	2	未知错误
	1	写保护擦除跳过，锁/解锁失败
	0	锁卡

表 11-28　第 3 种格式

字　节	位	含　义
1	7	开始位，始终为 0
	6	参数错误
	5	地址错误
	4	擦除序列错误
	3	CRC 错误
	2	非法命令
	1	擦除复位
	0	闲置状态
2 ~ 5	全部	操作条件寄存器，高位在前

写命令的例程如下：

```
//------------------------------------------------------------------------------
向 SD 卡中写入命令，并返回回应的第 2 个字节
//------------------------------------------------------------------------------
unsigned char Write_Command_SD(unsigned char *CMD)
{
unsigned char tmp;
unsigned char retry = 0;
unsigned char i;

//禁止 SD 卡片选
SPI_CS = 1;
//发送 8 个时钟信号
Write_Byte_SD(0xFF);
//使能 SD 卡片选
SPI_CS = 0;

//向 SD 卡发送 6 字节命令
for (i = 0;i < 0x06;i++)
{
Write_Byte_SD(*CMD++);
```

```
    }

    //获得 16 位的回应
    Read_Byte_SD( ); //读第 1 个字节,并忽略
    do
    {   //读取后 8 位
    tmp = Read_Byte_SD( );
    retry + + ;
    }
    while( ( tmp = = 0xff) && ( retry < 100) ) ;
    return( tmp) ;
}
```

（2）SD 卡的初始化

SD 卡的初始化是非常重要的，只有进行了正确的初始化，才能进行后面的各项操作。在初始化过程中，SPI 的时钟不能太快，否则会造成初始化失败。在初始化成功后，应尽量提高 SPI 的速率。在刚开始要先发送至少 74 个时钟信号，这是必须的。在很多读者的实验中，很多是因为疏忽了这一点，而使初始化不成功的。随后就是写入两个命令 CMD0 与 CMD1，使 SD 卡进入 SPI 模式。

初始化时序图如图 11-104 所示。

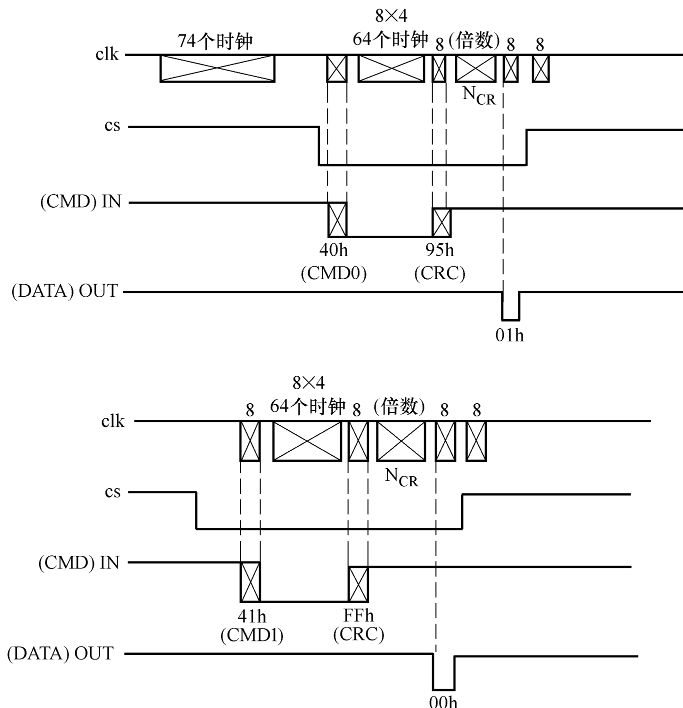


图 11-104　初始化时序图

初始化例程如下：

```
//----------------------------------------------------------------------
```

```
初始化 SD 卡到 SPI 模式
//---------------------------------------------------------------------
unsigned char SD_Init( )
{
   unsigned char retry,temp;
   unsigned char i;
   unsigned char CMD[ ] = {0x40,0x00,0x00,0x00,0x00,0x95};
SD_Port_Init( ); //初始化驱动端口

Init_Flag = 1; //将初始化标志置 1
for (i = 0;i < 0x0f;i + + )
{
   Write_Byte_SD(0xff); //发送至少 74 个时钟信号
}

//向 SD 卡发送 CMD0
retry = 0;
do
{ //为了能够成功写入 CMD0,在这里写 200 次
   temp = Write_Command_SD( CMD);
   retry + +;
   if( retry = = 200)
   { //超过 200 次
      return( INIT_CMD0_ERROR);//CMD0 错误
   }
}
while( temp! = 1);  //回应 01h,停止写入

//发送 CMD1 到 SD 卡
CMD[0] = 0x41; //CMD1
CMD[5] = 0xFF;
retry = 0;
do
{ //为了能成功写入 CMD1,写 100 次
   temp = Write_Command_SD( CMD);
   retry + +;
   if( retry = = 100)
   { //超过 100 次
      return( INIT_CMD1_ERROR);//CMD1 错误
   }
```

```
    }
    while(temp!  =0);//回应 00h 停止写入

    Init_Flag =0; //初始化完毕,初始化标志清零

    SPI_CS =1; //片选无效
    return(0); //初始化成功
}
```

4. SD 卡的寄存器与时序

SD 卡内部寄存器包括 CID、RCA、DSR、CSD、SCR 和 OCR 寄存器。其中 RCA 寄存器在 SPI 模式下不可用，DSR 为可选寄存器，OCR 为卡运行条件寄存器，描述卡的工作电压范围，它还包含一个上电状态标记位用于描述是否完成卡上电过程（驱动对卡在作初始化动作的时候特别要注意这点）。下面分别对 CID、CSD 和 SCR 这 3 个寄存器进行简单介绍。

（1）读取 CID 寄存器

CID 寄存器存储了 SD 卡的标识码。每一个卡都有唯一的标识码。CID 寄存器长度为 128 位。它的寄存器结构见表 11-29。

表 11-29　CID 寄存器结构

名　称	域	数据宽度	CID 划分
生产标识号	MID	8	[127:120]
OEM/应用标识	OID	16	[119:104]
产品名称	PNM	40	[103:64]
产品版本	PRV	8	[63:56]
产品序列号	PSN	32	[55:24]
保留	—	4	[23:20]
生产日期	MDT	12	[19:8]
CRC7 校验合	CRC	7	[7:1]
未使用,始终为 1	—	1	[0:0]

它的读取时序如图 11-105 所示。

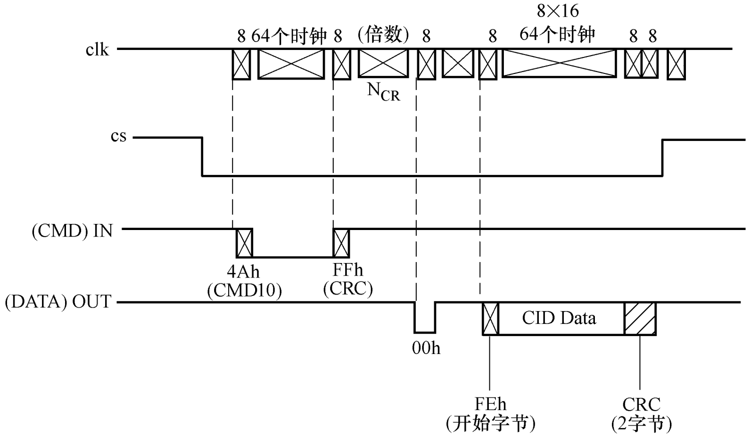


图 11-105　读取时序

由以上时序可知，用户在读取 CID 寄存器的软件参考代码如下：

```
//------------------------------------------------------------------------------
读取 SD 卡的 CID 寄存器 16 字节,成功返回 0
//------------------------------------------------------------------------------
unsigned char Read_CID_SD(unsigned char * Buffer)
{
  //读取 CID 寄存器的命令
  unsigned char CMD[] = {0x4A,0x00,0x00,0x00,0x00,0xFF};
  unsigned char temp;
  temp = SD_Read_Block(CMD,Buffer,16); //读 16 字节
  return(temp);
}
```

（2）读取 CSD 寄存器

CSD（Card-Specific Data）寄存器提供了读写 SD 卡的一些信息。其中的一些单元可以由用户重新编程。具体的 CSD 寄存器结构见表 11-30。

表 11-30　CSD 寄存器结构

名　　称	域	数据宽度	单元类型	CSD 划分
CSD 结构	CSD_STRUCTURE	2	R	[127:126]
保留	—	6	R	[125:120]
数据读取时间 1	TAAC	8	R	[119:112]
数据在 CLK 周期内读取时间 2(NSAC × 100)	NSAC	8	R	[111:104]
最大数据传输率	TRAN_SPEED	8	R	[103:96]
卡命令集合	CCC	12	R	[95:84]
最大读取数据块长	READ_BL_LEN	4	R	[83:80]
允许读的部分块	READ_BL_PARTIAL	1	R	[79:79]
非线写块	WRITE_BLK_MISALIGN	1	R	[78:78]
非线读块	READ_BLK_MISALIGN	1	R	[77:77]
DSR 条件	DSR_IMP	1	R	[76:76]
保留	—	2	R	[75:74]
设备容量	C_SIZE	12	R	[73:62]
最大读取电流(V_{DD} min)	VDD_R_CURR_MIN	3	R	[61:59]
最大读取电流(V_{DD} max)	VDD_R_CURR_MAX	3	R	[58:56]
最大写电流(V_{DD} min)	VDD_W_CURR_MIN	3	R	[55:53]
最大写电流(V_{DD} max)	VDD_W_CURR_MAX	3	R	[52:50]
设备容量乘子	C_SIZE_MULT	3	R	[49:47]
擦除单块使能	ERASE_BLK_EN	1	R	[46:46]
擦除扇区大小	SECTOR_SIZE	7	R	[45:39]
写保护群大小	WP_GRP_SIZE	7	R	[38:32]
写保护群使能	WP_GRP_ENABLE	1	R	[31:31]

（续）

名　　称	域	数据宽度	单元类型	CSD 划分
保留	—	2	R	[30:29]
写速度因子	R2W_FACTOR	3	R	[28:26]
最大写数据块长度	WRITE_BL_LEN	4	R	[25:22]
允许写的部分部	WRITE_BL_PARTIAL	1	R	[21:21]
保留	—	5	R	[20:16]
文件系统群	FILE_OFRMAT_GRP	1	R/W	[15:15]
复制标志	COPY	1	R/W	[14:14]
永久写保护	PERM_WRITE_PROTECT	1	R/W	[13:13]
暂时写保护	TMP_WRITE_PROTECT	1	R/W	[12:12]
文件系统	FIL_FORMAT	2	R/W	[11:10]
保留	—	2	R/W	[9:8]
CRC	CRC	7	R/W	[7:1]
未用,始终为 1	—	1		[0:0]

读取 CSD 寄存器的时序如图 11-106 所示。

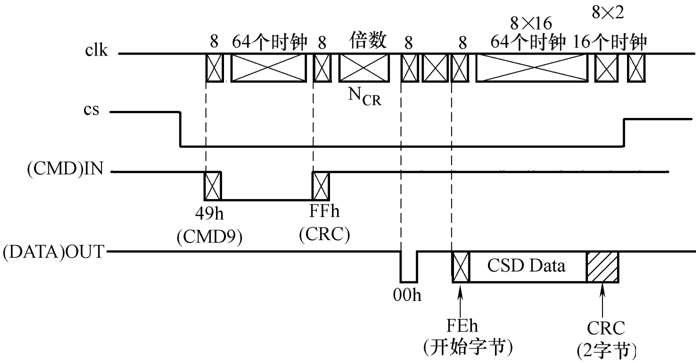


图 11-106　读取 CSD 寄存器的时序

相应的程序例程如下：

```
//------------------------------------------------------------------------------
读 SD 卡的 CSD 寄存器,共 16 字节,返回 0 说明读取成功
//------------------------------------------------------------------------------
unsigned char Read_CSD_SD(unsigned char *Buffer)
{
  //读取 CSD 寄存器的命令
  unsigned char CMD[] = {0x49,0x00,0x00,0x00,0x00,0xFF};
  unsigned char temp;
  temp = SD_Read_Block(CMD,Buffer,16); //读 16 字节
  return(temp);
```

```
}
```

(3) SCR 配置寄存器

SCR 寄存器结构见表 11-31。

表 11-31　SCR 寄存器结构

名　称	域	数据宽度	单元类型	CSD 划分
SCR 结构	SCR_STRUCTURE	4	R	[63:60]
SD 存储卡版本规格	SD_SPEC	4	R	[59:56]
擦除后数据状态	DATA_STAT_AFTER_ERASE	1	R	[55:55]
SD 卡安全性支持	SD_SECURITY	3	R	[54:52]
数据总线宽度支持	SD_BUS_WIDTHS	4	R	[61:48]
保留	—	16	R	[47:32]
保留(用于制造)	—	32	R	[31:0]

(4) 读取 SD 卡信息

综合上面对 CID 与 CSD 寄存器的读取，可以知道很多关于 SD 卡的信息，以下程序可以获取这些信息。

```
//-----------------------------------------------------------------------------
//返回
//  SD 卡的容量,单位为 MB
//  sector count and multiplier MB are in
u08 == C_SIZE / (2^(9-C_SIZE_MULT))
//  SD 卡的名称
//-----------------------------------------------------------------------------
void SD_get_volume_info()
{
   unsigned char i;
   unsigned char c_temp[5];
   VOLUME_INFO_TYPE SD_volume_Info, * vinf;
   vinf = &SD_volume_Info; //初始化指针;
//读取 CSD 寄存器
   Read_CSD_SD(sectorBuffer.dat);
//获取总扇区数
vinf->sector_count = sectorBuffer.dat[6] & 0x03;
vinf->sector_count <<= 8;
vinf->sector_count += sectorBuffer.dat[7];
vinf->sector_count <<= 2;
vinf->sector_count += (sectorBuffer.dat[8] & 0xc0) >> 6;
//获取倍数
vinf->sector_multiply = sectorBuffer.dat[9] & 0x03;
```

```
vinf->sector_multiply<<=1;
vinf->sector_multiply += (sectorBuffer.dat[10] & 0x80)>>7;
//获取 SD 卡的容量
vinf->size_MB=vinf->sector_count >> (9-vinf->sector_multiply);
//获取卡的名称
Read_CID_SD(sectorBuffer.dat);
vinf->name[0]=sectorBuffer.dat[3];
vinf->name[1]=sectorBuffer.dat[4];
vinf->name[2]=sectorBuffer.dat[5];
vinf->name[3]=sectorBuffer.dat[6];
vinf->name[4]=sectorBuffer.dat[7];
vinf->name[5]=0x00; //end flag
}
```

以上程序将信息装载到一个结构体中,这个结构体的定义如下:

```
typedef struct SD_VOLUME_INFO
{ //SD/SD Card info
  unsigned int size_MB;
  unsigned char sector_multiply;
  unsigned int sector_count;
  unsigned char name[6];
} VOLUME_INFO_TYPE;
```

(5) 扇区读

扇区读是对 SD 卡驱动的目的之一。SD 卡的每一个扇区中有 512 个字节，一次扇区读操作将把某一个扇区内的 512 个字节全部读出。过程很简单，先写入命令，在得到相应的回应后，开始数据读取。

扇区读的时序如图 11-107 所示。

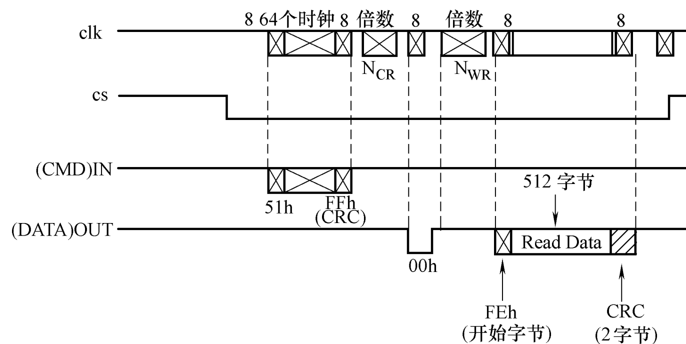


图 11-107 扇区读的时序

扇区读的程序例程如下:

```
unsigned char SD_Read_Sector(unsigned long sector,unsigned char *buffer)
```

```
{
    unsigned char retry;
    //命令 16
    unsigned char CMD[ ] = {0x51,0x00,0x00,0x00,0x00,0xFF};
    unsigned char temp;

    //地址变换,由逻辑块地址转为字节地址
    sector = sector≪ 9; //sector = sector * 512
    CMD[1] = ((sector & 0xFF000000) ≫24 );
    CMD[2] = ((sector & 0x00FF0000) ≫16 );
    CMD[3] = ((sector & 0x0000FF00) ≫8 );
    //将命令 16 写入 SD 卡
    retry = 0;
    do
    {  //为了保证写入命令,一共写 100 次
        temp = Write_Command_MMC(CMD);
        retry + +;
        if(retry = = 100)
        {
        return(READ_BLOCK_ERROR); //块写入错误
      }
    }
    while(temp! =0);

      //从 MMC/SD-Card 中读开始字节(FEh/开始字节)
      while (Read_Byte_MMC() ! = 0xfe);
      readPos = 0;
      SD_get_data(512,buffer) ;   //512 字节被读出到 buffer 中
    return 0;
    }
```

其中 SD_get_data 函数如下:

```
//------------------------------------------------------------------------
获取数据到 buffer 中
//------------------------------------------------------------------------
void SD_get_data(unsigned int Bytes,unsigned char * buffer)
{
    unsigned int j;
    for (j = 0;j < Bytes;j + +)
        * buffer + + = Read_Byte_SD();
}
```

（6）扇区写

扇区写是 SD 卡驱动的另一目的。每次扇区写操作将向 SD 卡的某个扇区中写入 512 个字节。过程与扇区读相似，只是数据的方向相反，与写入命令不同而已。

扇区写的时序如图 11-108 所示。

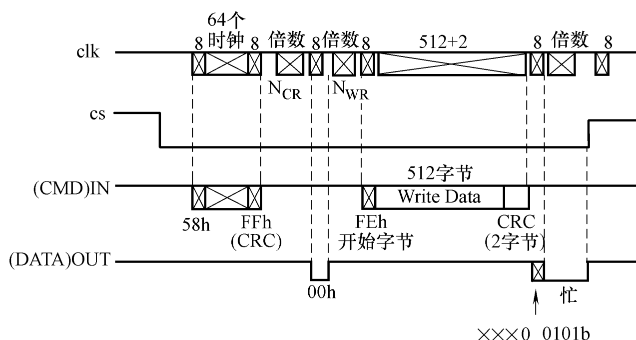


图 11-108　扇区写的时序

扇区写的程序例程如下：

```
//-----------------------------------------------------------------------------
写 512 个字节到 SD 卡的某一个扇区中去,返回 0 说明写入成功
//-----------------------------------------------------------------------------
unsigned char SD_write_sector(unsigned long addr,unsigned char * Buffer)
{
  unsigned char tmp,retry;
  unsigned int i;
  //命令 24
  unsigned char CMD[] = {0x58,0x00,0x00,0x00,0x00,0xFF};
  addr = addr<< 9; //addr = addr * 512

  CMD[1] = ((addr & 0xFF000000) >>24 );
  CMD[2] = ((addr & 0x00FF0000) >>16 );
  CMD[3] = ((addr & 0x0000FF00) >>8 );
  //写命令 24 到 SD 卡中去
  retry =0;
  do
  {  //为了可靠写入,写 100 次
    tmp = Write_Command_SD(CMD);
    retry + +;
    if(retry = =100)
    {
      return(tmp); //发送命令错误
```

```
        }
    }
    while(tmp! =0);

    //在写之前先产生 100 个时钟信号
    for (i=0;i<100;i++)
    {
        Read_Byte_SD();
    }

    //写入开始字节
    Write_Byte_MMC(0xFE);

    //现在可以写入 512 个字节
    for (i=0;i<512;i++)
    {
        Write_Byte_MMC(*Buffer++);
    }
    //CRC-Byte
    Write_Byte_MMC(0xFF); //Dummy CRC
    Write_Byte_MMC(0xFF); //CRC 码

    tmp=Read_Byte_MMC();  //读取响应
    if((tmp & 0x1F)! =0x05) // 写入的 512 个字节是未被接受
    {
        SPI_CS=1;
        return(WRITE_BLOCK_ERROR); //错误
    }
    //等到 SD 卡不忙为止
  //因为数据被接受后,SD 卡在向储存阵列中编程数据
    while (Read_Byte_MMC()! =0xff){};

    //禁止 SD 卡
    SPI_CS=1;
    return(0);//写入成功
  }
```

11.13.2 SD 卡读写应用实例

本例将对 SD 卡进行读写操作，验证结果通过串行口上传到电脑观察。

1. SD 卡 SPI 模式下与单片机的连接图（见图 11-109）

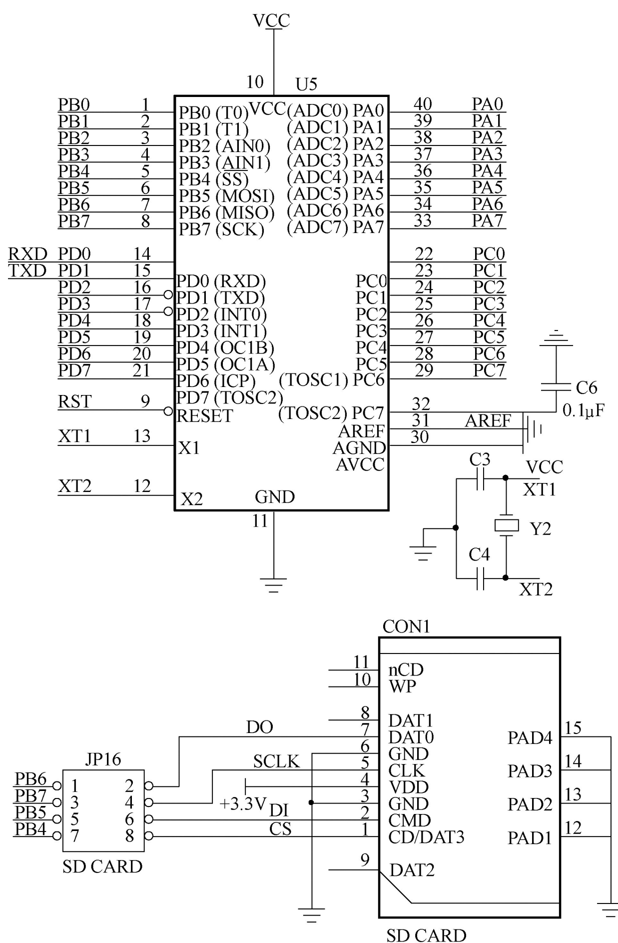


图 11-109　硬件原理图

2. 软件设计

```
/ ************************************************************ /
/ * SD 卡测试程序                                              * /
/ * 目标器件：ATmega16                                         * /
/ * 晶振：外部晶振 16MHz                                       * /
/ * 编译环境：ICCAVR 6.31A                                     * /
/ ************************************************************ /

/**************************包含头文件************************* /
#include  < iom32v. h >
#include  < macros. h >

/****************************端口定义************************* /
#defineSPI_DDR                 DDRB
```

```
#define SPI_PORT   PORTB
#define SPI_PIN         PINB
#define SPI_SS          4   //PB4
#define SPI_MOSI        5   //PB5
#define SPI_MISO        6   //PB6
#define SPI_SCK             7   //PB7

#define SET_SD_CS           PORTB |= (1<<4)
#define CLR_SD_CS           PORTB &= ~(1<<4)
#define SET_SPI_MOSIPORTB |= (1<<5)          //没数据时 SD_MI 应保持为高电平

/**************************定义全局变量***********************/
unsigned char buf[512];            //数据缓冲区
unsigned int k;

/***************************宏定义**************************/
#define   F_CPU   8000000          //设置外部晶振大小
#define   BAUD   19200                 //波特率设置

/***********************************************************
函数功能:延时子程序
入口参数:k
出口参数:
***********************************************************/
void delay_1ms(void)              //1ms 延时函数
  {
    unsigned int i;
    for (i=0;i<1840;i++);
  }
//ms 级别是对的,在 8MHz 的情况
void delay_nms(unsigned int n)   //Nms 延时函数
  {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1ms();
  }

/***********************************************************
函数功能:uart 初始化程序
入口参数:
```

```
出口参数:
*********************************************************/
void USART_Init(void)
{
    /*设置波特率*/
    //UBRRH = 0;
    //UBRRL = 25;/*设置波特率,16MHz 晶振,38400 波特率,参考 DataSheet */
    UBRRH = (((F_CPU/BAUD)/16)-1)/256;
    UBRRL = (((F_CPU/BAUD)/16)-1)%256;

    UCSRB |= (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);
    //使能发送,接收,接收完成中断
}

/*********************************************************
函数功能:uart 发送单字节数据
入口参数:c
出口参数:
*********************************************************/
void USART_Send_Char(unsigned char data)
{
    /*等待发送缓冲器为空*/
    while( !(UCSRA & (1<<UDRE)));
    /*将数据放入缓冲区,发送数据*/
    UDR = data;
}

/*********************************************************
函数功能:uart 发送字符串数据
入口参数:*s
出口参数:
*********************************************************/
void USART_Send_Str(unsigned char *str)
{
    while(*str!='\0')
    {
      USART_Send_Char(*str);
      str++;
    }
}
```

```
/*********************************************************
函数功能:SPI 低速模式
入口参数:
出口参数:
*********************************************************/
void SPI_Low(void)
{
    SPCR =0;
    SPCR =(1 < <  SPE) | (1 < <MSTR) | (1 < <SPR0) | (1 < <SPR1);
    //使能 SPI,主机方式,MSB 在前,模式 0,128 分频
}

/*********************************************************
函数功能:SPI 高速模式
入口参数:
出口参数:
*********************************************************/
void SPI_High(void)
{
    SPCR =0;
    SPCR =(1 < <SPE) | (1 < <MSTR);
    SPSR |= (1 < <SPI2X);
    //使能 SPI,主机方式,MSB 在前,模式 0,4 分频,2 倍频
}

/*********************************************************
函数功能:SPI 初始化
入口参数:
出口参数:
*********************************************************/
void SPI_Init(void)
{
    SPI_PORT =(1 < <SPI_SS) | (1 < <SPI_MISO);      //将 SS 置位输出拉高,MISO
                                                    输入带上拉
    SPI_DDR =(1 < <SPI_SS) | (1 < <SPI_MOSI) | (1 < <SPI_SCK);
    //将 SS SCK MOSI 置为输出
}

/*********************************************************
函数功能:SPI 读写
```

```
入口参数:
出口参数:
*************************************************************/
unsigned char SPI_RW(unsigned char dat)//SPI 读写 1 字节
{
    SPDR = dat;
    while(!(SPSR & (1<<SPIF)));
    return (SPDR);
}

/*************************************************************
函数功能:SPI 写命令
入口参数:
出口参数:
*************************************************************/
unsigned char SD_Write_Com(unsigned char *cmd)
{
    unsigned char i,temp,retry;

    SET_SD_CS;              //关片选
    SPI_RW(0XFF);           //提高兼容性,如果没有这里,有些 SD 卡可能不支持
    CLR_SD_CS;              //开片选(后面的读写扇区可以省去开片选)
    asm("nop");

    for(i=0;i<6;i++)
    {
        SPI_RW(*cmd);
        cmd++;
    }

    SPI_RW(0XFF);           //写入指令后附加 8 个时钟脉冲

    retry=0;
    do                      //不断地读
    {
        temp=SPI_RW(0XFF);
        retry++;
    }while((temp==0XFF) && (retry<254));

    return (temp);
```

```
}

/****************************************************************
函数功能:SD 卡初始化
入口参数:
出口参数:
****************************************************************/
unsigned char SD_Init(void)
{
    unsigned char i,temp;
    unsigned int retry;          //重试次数 16 位数据(适应大容量 SD 卡)
    unsigned char CMD[] = {0x40,0x00,0x00,0x00,0x00,0x95};

    SPI_Init();
    SPI_Low();                   //SPI 初始化低速模式

    for(i = 0;i < 200;i++)  //等待 MMC/SD 准备...
    {
        asm("nop");
    }

    retry = 0;
    do
    {
        SET_SD_CS;
        for (i = 0;i < 0x0f;i++)
        {
            SPI_RW(0xff);   //至少 74 个信号脉冲
        }

        temp = SD_Write_Com(CMD);   //写入 CMD0 号命令,最大重试次数 254
        retry++;
        if(retry == 600)
        {
          SET_SD_CS;
          return(1);          //写入 CMD0 号命令错误
        }
    }while(temp != 1);

    CMD[0] = 0x41;            //CMD1 号命令
```

```
    CMD[5] =0xFF;

    retry =0;
    do
    {
        temp = SD_Write_Com(CMD);
        retry + +;
        if(retry >60000)    //写入 CMD1 号命令,最大重试次数 60000
        {
            SET_SD_CS;  //关闭片选
            return(1);      //写 CMD1 命令失败
        }
    }while(temp != 0);
    SET_SD_CS;              //关闭片选

    SPI_High();             //初始化成功,SPI 高速模式
    SET_SPI_MOSI;           //没数据时 SD_MI 应保持为高电平
    return(0);              //成功初始化
}
/************************************************************
函数功能:SD 卡写一个扇区
入口参数:
出口参数:
************************************************************/
unsigned char   SD_Write_Sector(unsigned long addr,unsigned char *buffer)
{//向 SD 卡中的指定地址的扇区写入 512 个字节,使用 CMD24(24 号命令,单块写命令)
    unsigned char temp,retry;
    unsigned int i;
    unsigned char cmd24[] = {0x58,0x00,0x00,0x00,0x00,0xff};

    addr < < = 9; //将块地址(扇区地址)转为字节地址(所以 SD 卡的最大容量为 4GB)

    cmd24[1] =((addr & 0xFF000000) > > 24);
    cmd24[2] =((addr & 0x00FF0000) > > 16);
    cmd24[3] =((addr & 0x0000FF00) > > 8);
    cmd24[4] =((addr & 0x000000FF) > > 0);//可以省去

    retry =0;
    do
    {
```

```
        temp = SD_Write_Com(cmd24);
        retry++;
        if(retry > 200)
        {
            return(1);              //CMD24 命令写入失败
        }
    }while(temp != 0);

    SPI_RW(0XFF);
    SPI_RW(0XFF);
    SPI_RW(0XFF);                   //按照时序要求插入若干时序信号

    SPI_RW(0XFE);                   //写入开始字节 0xfe,后面就是要写入的 512 个字
                                      节的数据

    for (i=0;i<512;i++)             //将缓冲区中要写入的 512 个字节写入 SD 卡
    {
        SPI_RW(buffer[i]);
    }

    SPI_RW(0xFF);
    SPI_RW(0xFF);                   //两个字节的 CRC 校验码,不用关心

    temp = SPI_RW(0XFF);            //读取返回值
    if((temp & 0x1F) != 0x05)       //如果返回值是 XXX0 0101b 说明数据已经被 SD
                                      卡接受了
    {
        SET_SD_CS;
        return(1);                  //写块数据失败
    }

    while(SPI_RW(0XFF) != 0xff);
    //等待 SD 卡不忙
    //等到 SD 卡不忙(数据被接受以后,SD 卡要将这些数据写入到自身的 FLASH 中,
      需要一段时间)
    //忙时,读回来的值为 0x00,不忙时,为 0xff

    SET_SD_CS;                      //关闭片选

    SPI_RW(0XFF);                   //按照 SD 卡的操作时序在这里补 8 个时钟
```

```
    SET_SPI_MOSI;                    //没数据时 SD_MI 应保持为高电平
    return(0);                       //返回 0,说明写扇区操作成功
}
/*************************************************************
函数功能:SD 卡读一个扇区
入口参数:
出口参数:
*************************************************************/
unsigned char  SD_Read_Sector(unsigned long addr,unsigned char *buffer)
{//从 SD 卡的指定扇区中读出 512 个字节,使用 CMD17(17 号命令)
    unsigned long i;
    unsigned char retry,temp;
    unsigned char cmd17[] = {0x51,0x00,0x00,0x00,0x00,0xFF}; //CMD17 的字节序列

    addr <<= 9;      //将块地址(扇区地址)转为字节地址

    cmd17[1] = ((addr & 0xFF000000) >> 24);
    cmd17[2] = ((addr & 0x00FF0000) >> 16);
    cmd17[3] = ((addr & 0x0000FF00) >> 8);
    cmd17[4] = ((addr & 0x000000FF) >> 0);//可以省去

    retry = 0;
    do
    {
        temp = SD_Write_Com(cmd17);        //写入 CMD17
        retry++;
        if(retry > 250)                   //最大重试次数 250
        {
            SET_SD_CS;
            return(1);                    //读块失败
        }
    }while(temp != 0);

    while (SPI_RW(0XFF) != 0xfe);       //一直读,当读到 0xfe 时,说明后面的是
                                          512 字节的数据了

    for(i = 0;i < 512;i++)           //将数据写入到数据缓冲区中
```

```
        {
            buffer[i] = SPI_RW(0XFF);
        }

        SPI_RW(0XFE);
        SPI_RW(0XFE);                //读取两个字节的 CRC 校验码,不用关心它们

        SET_SD_CS;                   //SD 卡关闭片选

        SPI_RW(0xFF);                //按照 SD 卡的操作时序在这里补 8 个时钟

        SET_SPI_MOSI;                //没数据时 SD_MI 应保持为高电平
        return 0;                    //返回 0,说明读扇区成功
    }

    /*********************************************************
    函数功能:主程序
    入口参数:
    出口参数:
    *********************************************************/
    int main(void)
    {
        delay_nms(50);
        PORTB |= (1<<4);
        DDRB |= (1<<4);

        for(k = 0;k < 512;k++)
        {
            buf[k] = k;
        }

        USART_Init();
        USART_Send_Str("SD Card Testing Demo\r\n");

        if(SD_Init() == 0)
            USART_Send_Str("SD Card Init OK!\r\n");
        else
            USART_Send_Str("SD Card Init Fault!\r\n");
```

```
    //写 512 字节
    f(SD_Write_Sector(70,buf) == 0)
    {
        USART_Send_Str("SD Card Write OK!\r\n");
        for(k=0;k<512;k++)
        {
            USART_Send_Char(buf[k]/100+0x30);
            USART_Send_Char(buf[k]/10%10+0x30);
            USART_Send_Char(buf[k]%10+0x30);
            //USART_Send_Str("\r\n");
        }
        USART_Send_Str("\r\n");
    }
    else
        USART_Send_Str("SD Card Write Fault!\r\n");
    for(k=0;k<512;k++)                //清零缓冲区
    {
        buf[k]=0;
    }
    //读 512 字节
    if(SD_Read_Sector(70,buf) == 0)
    {
        USART_Send_Str("SD Card Read OK!\r\n");
        for(k=0;k<512;k++)
        {
            USART_Send_Char(buf[k]/100+0x30);
            USART_Send_Char(buf[k]/10%10+0x30);
            USART_Send_Char(buf[k]%10+0x30);
            //USART_Send_Str(\r\n");
        }
        USART_Send_Str("\r\n");
    }
    else
        USART_Send_Str("SD Card Read Fault!\r\n");

    while(1)
    {
        asm("nop");
    }
}
```

11.14　LED 点阵显示屏的应用实例

LED 是发光二极管英文（Light Emitting Diode）的简称。LED 具有高亮度、视觉明显、图像清晰、色彩多样、可靠性好、功耗低、易于集成、驱动简单易于集成、驱动简单、光效高、寿命长等优点而成为众多显示界面以及户外媒体显示的理想选择。利用 LED 器件制造大型平板显示屏系统成为 LED 器件应用的重要领域，进而推动了大型显示屏系统发展并形成一种产业。用 LED 器件组成显示屏的最大特点在于其制造不受面积限制，可以达到几十甚至几百平方米以上，应用于室内外的各种公共场合显示文字、图形、图像、动画、视频等各种信息，其发展前景极为广阔。

11.14.1　LED 点阵的种类及结构

LED 点阵就是 LED 以矩阵形式布置而得名。LED 点阵通常有 4×4、4×8、5×7、5×8、8×8、16×16、24×24、40×40 等形式；根据像素的数目分为单基色、双基色、三基色等。单基色点阵顾名思义只能显示固定色彩，如红、绿、黄等单色，双基色和三基色点阵显示内容的颜色由像素内不同颜色发光二极管点亮组合方式决定，如红绿都亮时可显示黄色，如果按照脉冲方式控制二极管的点亮时间，则可实现 256 或更高级灰度显示，即可实现真彩色显示。根据像素颜色的不同所显示不同颜色的文字、图像等内容。图 11-110 为 8×8LED 点阵外观及引脚图，有序控制某些 LED 亮与灭，就可以显示出数字、字母、文字等图案。

11.14.2　8×8 单色点阵 LED 的工作原理

单色 LED 点阵一般只是控制点阵中各发光器件的通断（发光或熄灭），而不控制 LED 的发光强弱。如图 11-111 所示，8×8 点阵由 64 个发光二极管组成，每个发光二极管放置在行线和列线的交叉点上。单只 LED 的正向工作电压（V_F）=1.8V，正向静态工作电流（I_F）=8～20mA，动态瞬间电流可达 80～160mA。当对应的某一列置低电平，某一行置高电平时，则该列与该行交叉的二极管点亮。

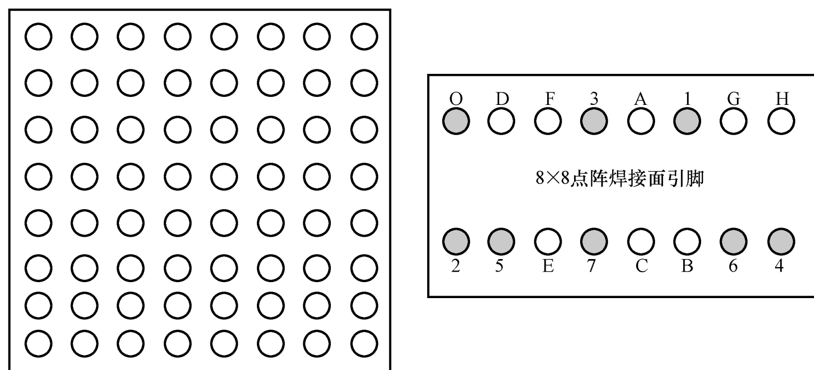


图 11-110　8×8LED 点阵外观及引脚图

下面介绍一种实用的简易汉字显示屏的制作，实际使用时可根据这个原理自行扩充显示。我们知道在 UCDOS 中文宋体字库中，每一个字由 16 行 16 列的点阵组成显示，即每一个汉字由 256 点阵来表示。以汉字“大”为例，如图 11-112 所示。

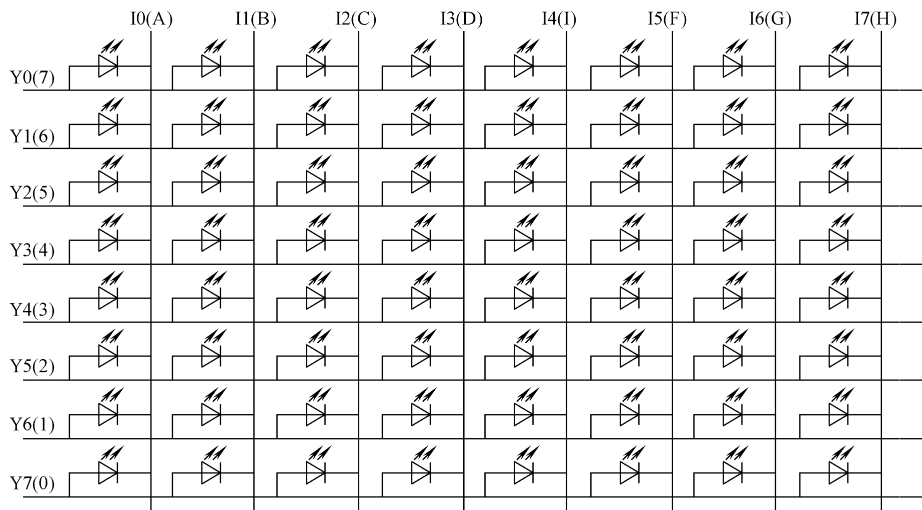


图 11-111　8 ×8 点阵 LED 等效电路

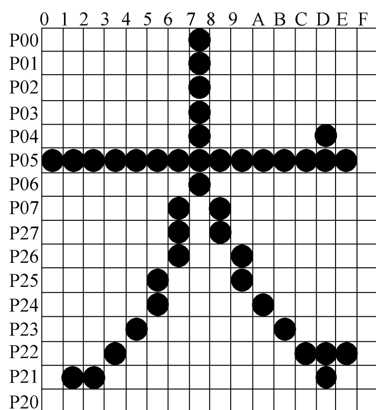


图 11-112　“大”点阵显示图

一般把它拆分为左部和右部，左部由 16 ×8 点阵组成，右部也由 16 ×8 点阵组成。图中黑点表示 LED 需要点亮，先扫描 P00 行的左半行，假设 LED 点亮为 1，熄灭为 0，则 P00 行的左半行可表示为 00000001，转化为十六进制可表示为 01H；同理，右半行为 00000000，转化为十六进制可表示为 00H；依次下去，继续进行下面的扫描，共扫描 32 个 8 位，可以得出汉字“大”。“大”的扫描代码为：P00：01H，00H；P01：01H，00H；P02：01H，00H；P03：01H，00H；P04：01H，04H；P05：0xffH，0xfeH；P06：01H，00H；P07：02H，80H；P08：02H，80H；P09：02H，40H；P10：04H，40H；P11：04H，10H；P12：80H，10H；P13：10H，0eH；P14：60H，04H；P15：00H，00H；

对于其他种类的点阵，无论显示何种字体或图像，都可以用这个方法来分析出它的扫描代码，从而显示在屏幕上。

11.14.3　LED 点阵显示屏系统设计

LED 显示屏硬件系统包括驱动部分和控制部分，本例以显示字母“F”为例，介绍其软硬件设计。

1. 硬件原理图（见图 11-113）

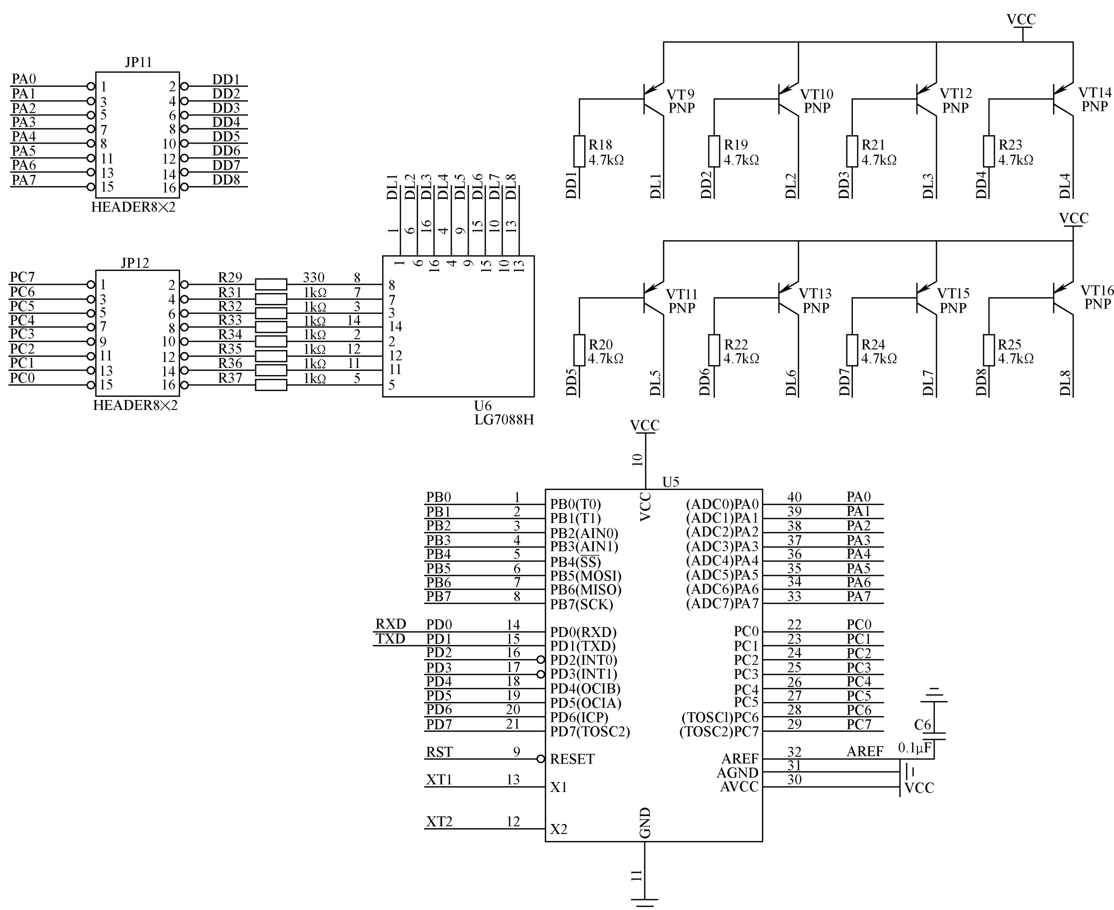


图 11-113　硬件原理图

2. 软件设计

```
/***********************************************************/
/* LED 点阵测试程序                                          */
/* 目标器件:ATmega16                                         */
/* 晶振:RC 1MHz                                              */
/* 编译环境:ICCAVR 6.31A                                     */
/***********************************************************/

/*************************包含头文件*************************/
#include <iom16v.h>
#include <macros.h>

extern const unsigned char tab[8]={0xff,0xc3,0xdf,0xc3,0xdf,0xdf,0xdf,0xff};
```

```
/*****************************************************
函数功能:延时程序
入口参数:
出口参数:
*****************************************************/
void delay_1us(void)                    //1μs 延时函数
  {
  asm("nop");
  }

void delay_nus(unsigned int n)          //Nμs 延时函数
  {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1us();
  }

void delay_1ms(void)                    //1ms 延时函数
  {
    unsigned int i;
    for (i=0;i<140;i++);
  }

void delay_nms(unsigned int n)          //Nms 延时函数
  {
    unsigned int i=0;
    for (i=0;i<n;i++)
    delay_1ms();
  }

/*****************************************************
函数功能:显示子程序
入口参数:k
出口参数:
*****************************************************/
void display()
{
    unsigned char i,j;
    for(i=0;i<8;i++)
    {
```

```
        j = 7 - i;
        PORTA& = ~(1<<j);
        PORTC = tab[i];
        delay_nms(2);
        PORTA = 0xff;
    }

}

/**************************************************************
函数功能:主程序
入口参数:
出口参数:
**************************************************************/
void main(void)
{
    DDRC = 0xff;
    DDRA = 0xff;

    while(1)
    {
        display();//显示“F”字样
    }
}
```

参 考 文 献

［1］ 徐玮，徐富军，沈建良．C51 单片机高效入门［M］．北京：机械工业出版社，2007.

［2］ 徐玮，沈建良．单片机快速入门［M］．北京：北京航空航天大学出版社，2008.

［3］ 徐玮，沈建良．PIC 单片机快速入门［M］．北京：北京航空航天大学出版社，2010.

［4］ 张志良．单片机原理与控制技术［M］．北京：机械工业出版社，2001.

［5］ 吴金戌，沈庆阳，郭庭吉．8051 单片机实践与应用［M］．北京：清华大学出版社，2002.

［6］ 谭浩强．C 程序设计［M］．北京：清华大学出版社，1991.

［7］ 马忠梅，籍顺心，张凯，等．单片机的 C 语言应用程序设计［M］．3 版．北京：北京航空航天大学出版社，2003.

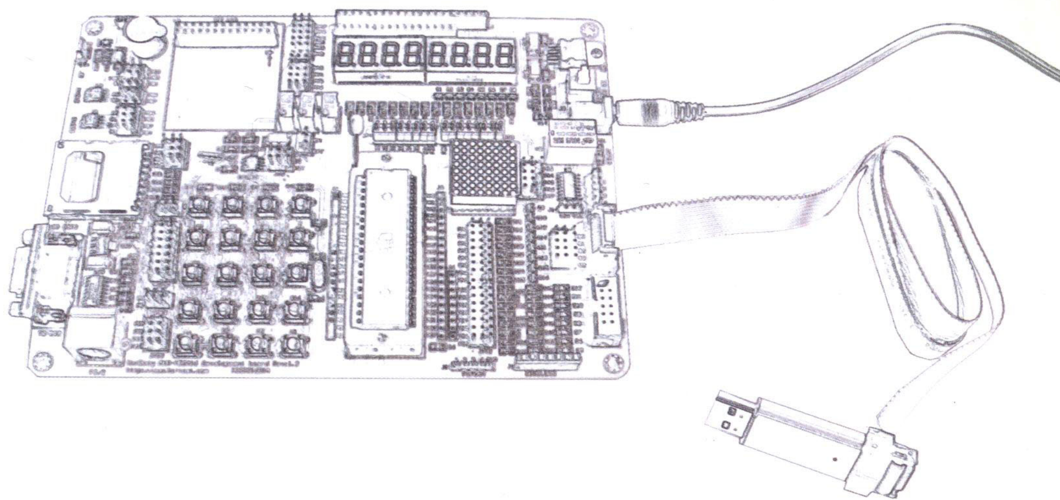
［8］ 马潮．AVR 单片机嵌入式系统原理与应用实践［M］．北京：北京航空航天大学出版社，2007.

［9］ 沈文，詹卫前．AVR 单片机 C 语言开发入门指导［M］．北京：清华大学出版社，2003.

［10］ 金钟夫．AVR ATmega128 单片机 C 程序设计与实践［M］．北京：北京航空航天大学出版社，2007.

［11］ 刘海成．AVR 单片机原理及测控工程应用：基于 ATmega48/ATmega16［M］．北京：北京航空航天大学出版社，2008.

［12］ 耿德根．AVR 高速嵌入式单片机原理与应用（修订版）［M］．北京：北京航空航天大学出版社，2003.

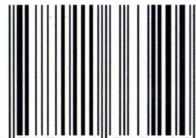


ISBN 978-7-111-36320-0
ISBN 978-7-89433-236-3(光盘)
策划编辑：林春泉
封面设计：路恩中

地址：北京市百万庄大街22号
电话服务
社服务中心：(010)88361066
销售一部：(010)68326294
销售二部：(010)88379649
读者购书热线：(010)88379203
邮政编码：100037
网络服务
门户网：http://www.cmpbook.com
教材网：http://www.cmpedu.com
封面无防伪标均为盗版

上架指导：工业技术／通信技术

ISBN 978-7-111-36320-0



定价：68.00元（含1CD）